HT-IDE3000 使用手册

本使用手册版权为盛群半导体股份有限公司所有，非经盛群半导体股份有限公司书面授权同意，不得通过任何形式复制、储存或传输。

注意

使用指南中所出现的信息在出版当时相信是正确的，然而盛群对于说明书的使用不负任何责任。文中提到的应用目的仅仅是用来做说明，盛群不保证或表示这些没有进一步修改的应用将是适当的，也不推荐它的产品使用在会由于故障或其它原因可能会对人身造成危害的地方。盛群产品不授权使用于救生、维生器件或系统中做为关键器件。盛群拥有不事先通知而修改产品的权利。对于最新的信息，请参考我们的网址 http://www.holtek.com.tw

目录

第一部份

集成开发环境

第一章

概要与安装

1

为了简化应用程序的开发过程，支持工具的重要性和有效性对于单片机来说是不可低估的。为了支持所有系列的单片机，盛群用心的提供具有完整功能的工具，让用户在开发与使用上更加便利。例如众所周知的 HT-IDE 集成开发环境，软件方面具有 HT-IDE3000，提供友善的窗口介面以便进行程序的编辑及除错，同时硬件方面有 HT-ICE 仿真器，它提供了多种实时仿真功能，包括多功能跟踪、单步执行以及设定断点功能。HT-IDE 开发系统提供完整系列的接口卡并定期的更新服务软件包，因此保证设计者可以有最佳的工具，且能以最高的效率进行单片机应用程序的设计与开发。

HT-IDE 集成开发环境

HT-IDE (Holtek Integrated Development Environment)是一个具有高效能，用于设计盛群 8 位单片机应用程序的集成开发环境。系统中的硬件及软件工具能够帮助客户使用盛群 8 位单片机芯片，快速且容易地开发应用程序。在 HT-IDE 集成开发环境中，最主要的组件之一为 HT-ICE（In-Circuit Emulator），它提供了盛群 8 位单片机的实时仿真功能以及强而有力的除错与跟踪功能。最新版的 HT-ICE 仿真器更进一步将 OTP writer 烧录器整合在仿真器上，提供给使用者从程序设计、除错到烧录的所有的功能。

在软件方面，HT-IDE3000 开发系统提供友善的工作平台。将所有的软件工具，例如编辑器、编译器、连接器、函数库管理器和符号除错器，集成为一个窗口环境，使程序开发过程更为容易。HT-IDE3000 软件还提供软件仿真器功能，无需接上 HT-ICE 仿真器，即可进行程序开发。此软件仿真器可以仿真盛群 8 位单片机，以及所有 HT-ICE 仿真器的基本功能。

HT-IDE3000 使用手册包含 HT-IDE 开发系统的相关细节。为了提供 HT-IDE3000 的安装作业以及确保开发系统包含有最新的单片机和软件更新信息，盛群也定期提供 HT-IDE3000 服务软件包(Service Pack)。这些服务软件包不是用来取代 HT-IDE3000，它必须要在 HT-IDE3000 系统软件已安装后才能被安装。

HT-IDE3000 具有下列特性：

→ **仿真**

- 程序指令的实时仿真

→ **硬件**

- 使用及安装容易
- 可使用内部或外部振荡器
- 断点功能
- 跟踪功能，跟踪仿真器提供触发条件
- HT-ICE 通过打印口与计算机连接
- 使用者的应用电路板通过 I/O 接口卡连接至 HT-ICE
- HT-ICE 中集成 OTP 烧录器

→ **软件**

- 窗口结构的软件工具
- 源程序层次的除错器（符号除错器）
- 支持多个源程序文件的工作平台（一个应用项目可以包含一个以上的源程序文件）
- 所有工具被使用于开发、除错、评估和生成最终的应用程序代码（Mask ROM file）
- 可将公用程序建立成函数库并在以后连接到其它的项目中
- 软件仿真器不需要连接 HT-ICE 硬件即可进行程序的仿真和除错
- 虚拟外围器件管理（VPM）可以仿真外围器件的行为
- LCD 仿真器可以仿真 LCD 面板的动作

盛群单片机仿真器(HT-ICE)

对于盛群的 8 位单片机而言，盛群的 ICE 是全功能的仿真器，系统中的硬件及软件工具能帮助客户快速且容易的开发应用程序。系统中最主要的是硬件仿真器，除了能有效地提供除错和跟踪功能之外，还能以实时的方式进行盛群 8 位单片机的仿真工作。在软件方面，HT-IDE3000 开发系统提供友善的工作平台，将所有软件工具，例如编辑器、编译器、连接器、函数库管理器和符号除错器，整合到窗口环境。此外系统能在软件仿真模式下，不需连接 HT-ICE 硬件即可执行程序的仿真。

HT-ICE 接口卡

HT-ICE 提供的接口卡(图 1-1)可以被许多应用电路使用，但使用者也可自行设计接口卡。将必要的接口电路放在他们自己的接口卡上，使用者可以直接把他们的应用电路板连接到 HT-ICE 的 CN1 和 CN2 连接器。

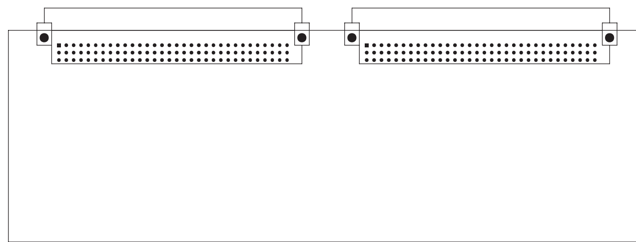


图 1-1

OTP 烧录器

所有盛群的 OTP 芯片都有烧录器支持。对于工业级的 OTP 芯片烧录而言，盛群 OTP writer 工具提供一个快速且有效的方法，来进行 OTP 小规模的量产烧录。最新版的 HT-ICE 仿真器更进一步将 OTP writer 整合在 HT-ICE 仿真器上，提供使用者从程序设计、除错到烧录验证等所有必备的功能。另外有更多的烧录器供应商可提供高效率及高容量的烧录服务。请参阅网站以获得进一步供应商信息。

OTP 适配卡

OTP 烧录器本身提供一个标准的芯片插座，而 OTP 适配卡则是使用在烧录其它封装形式的 OTP 芯片，这些封装形式的芯片无法在标准芯片插座上烧录，需要在 OTP 烧录器插上此适配卡才能烧录。

系统配置

HT-IDE3000 系统配置如图 1-2，主机需为 Pentium 兼容机器，软件为 Windows 95/98/NT/2000/XP 或更新，注意在 Windows NT/2000/XP 系统下安装 HT-IDE3000 时，需要在 supervisor privilege 模式下执行。

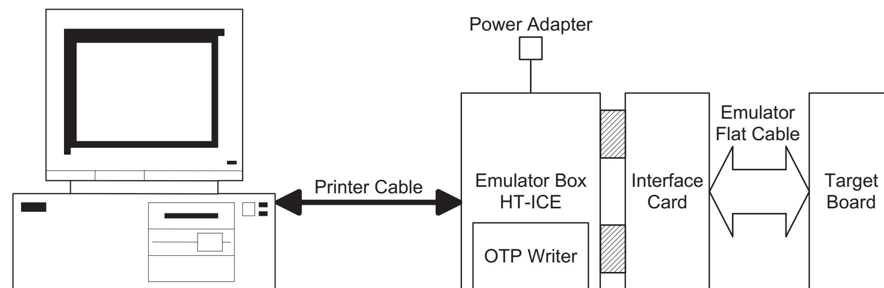


图 1-2

→ **HT-IDE3000 系统包含下列硬件组成**

- HT-ICE 仿真器，包含印刷电路板（PCB），其中有一个打印机端口的连接头，可与个人计算机连接，输入/输出信号接头及 LED 电源指示灯，如图 1-3。
- 连接目标电路板与 HT-ICE 仿真器的 I/O 接口卡
- 变压器(输出 16V)
- D 型 25 脚打印机并口线
- OTP writer 烧录器

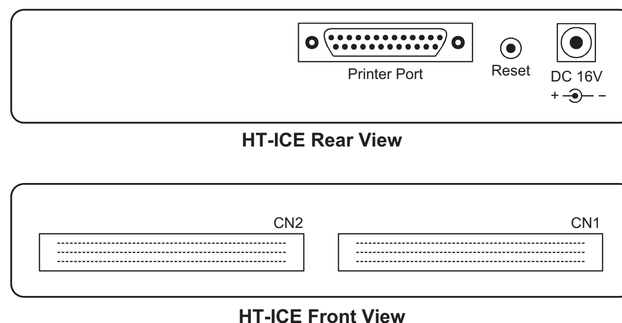


图 1-3

安装

系统需求

安装 HT-IDE3000 系统的硬件及软件需求如下：

- Pentium 等级以上 CPU 的 PC/AT 兼容机器
- SVGA 彩色显示器
- 至少 32M 以上的 RAM
- CD ROM 装置(从 CD 安装时需要)
- 至少 20M 以上的硬盘空间
- 具有并行口，可连接 PC 和 HT-ICE
- Windows 95/98/NT/2000/XP 操作系统

* Window 95/98/NT/2000/XP 是 Microsoft 公司注册商标

硬件安装

- 步骤 1
  将电源变压器插入 HT-ICE 的电源插孔
- 步骤 2
  通过 I/O 接口卡或排线连接目标电路板与 HT-ICE
- 步骤 3
  使用打印并口线连接 HT-ICE 与主机

此时 HT-ICE 上的 LED 应该是亮的，如果不是，则重新操作连接的步骤或与代理商联系。

警告: 请小心使用电源变压器，勿使用输出不是 16V 的变压器，否则可能导致 HT-ICE 损坏。因此强烈建议使用由盛群所提供的变压器。首先将电源变压器插入 HT-ICE 的电源插座。

软件安装

- 步骤 1

  将 HT-IDE3000 CD 放入 CD ROM 装置中，将出现下列的对话框。

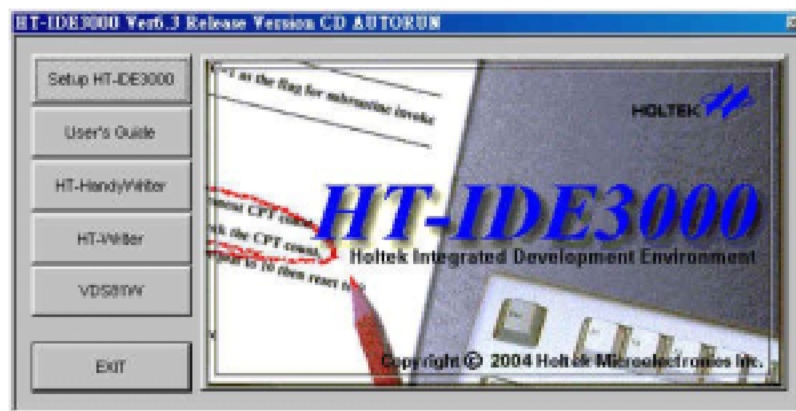


图 1-4

按下<HT-IDE3000>按钮，下列的对话框会出现。

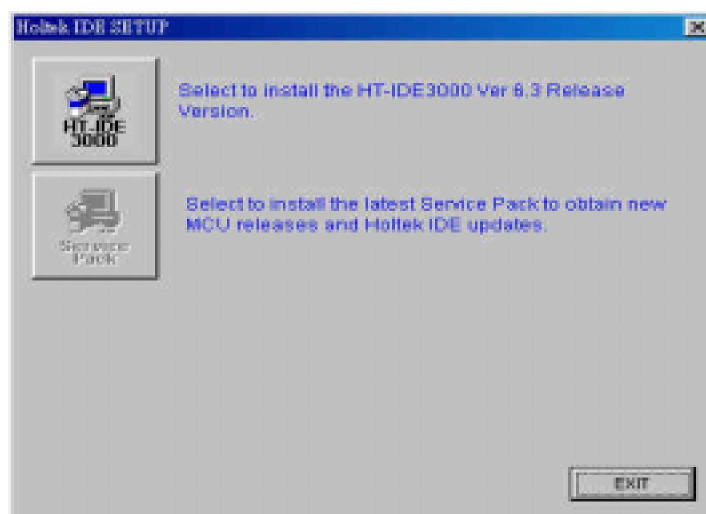


图 1-5

依照你想要安装的功能，按下<HT-IDE3000>或<Service Pack>按钮。如果是首次安装，请按<HT-IDE3000>，如果已安装 HT-IDE3000，而需要更新版本时，请按<Service Pack>。

以下为选择安装<HT-IDE3000>的范例说明，按下<HT-IDE3000>。

- 步骤 2

  按下<Next>按钮(继续安装)或按下<Cancel>按钮(中止安装)。

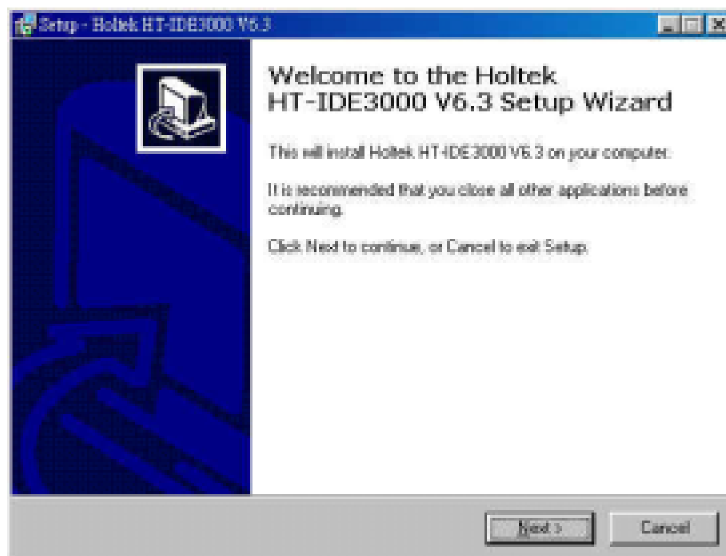


图 1-6

- 步骤 3

  下面显示的对话框会要求使用者输入安装路径。

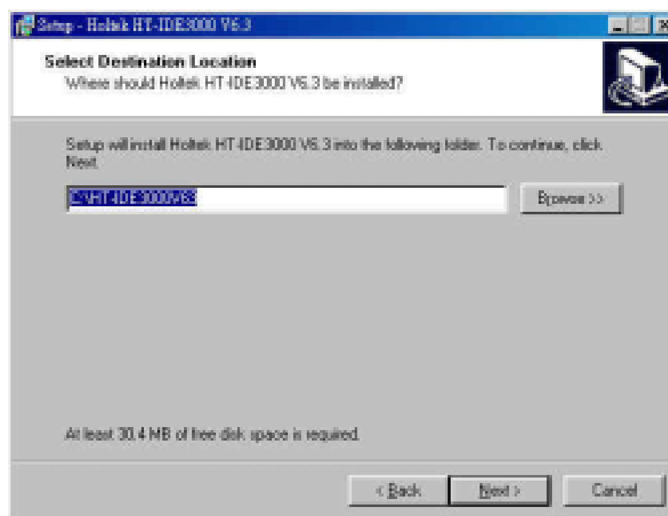


图 1-7

- 步骤 4

  指定你希望安装 HT-IDE3000 的路径，然后按下<Next>按钮。

- 步骤 5
  SETUP 会将所有文件复制到你指定的路径下。

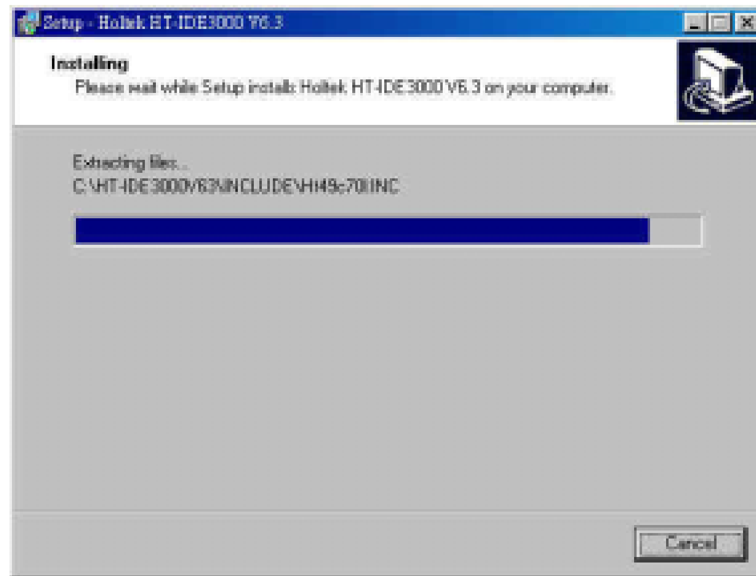


图 1-8

- 步骤 6
  安装成功的话，则会出现下面的对话框。

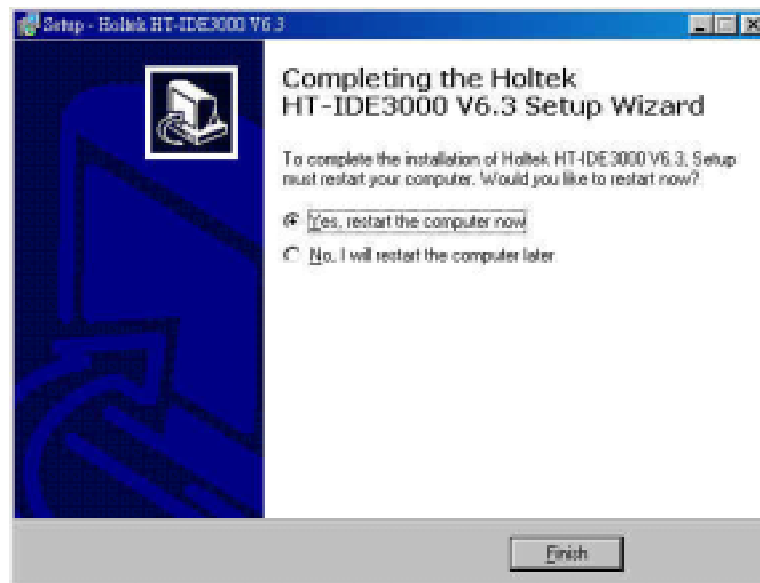


图 1-9

- 步骤 7

  按下<Finish>按钮完成安装并重新启动计算机系统，接下来你就可以执行 HT-IDE3000 程序了。SETUP 会在你指定的路径下，建立四个子文件夹(BIN、INCLUDE、LIB、SAMPLE)。BIN 子文件夹包含所有的系统可执行文件(EXE)、动态连接库(DLL)和配置文件(CFG、FMT)，INCLUDE 子文件夹包含所有由盛群所提供的包含文件(.H、.INC)，LIB 子文件夹包含由盛群所提供的函数库文件(.LIB)，SAMPLE 子文件夹包含范例程序。

  注意，在第一次执行 HT-IDE3000 之前，系统会要求输入如图 1-10 所示的公司资料，请选择适当的区域并填入公司名称及识别码，其中识别码可由 HT-IDE3000 的供应商提供。

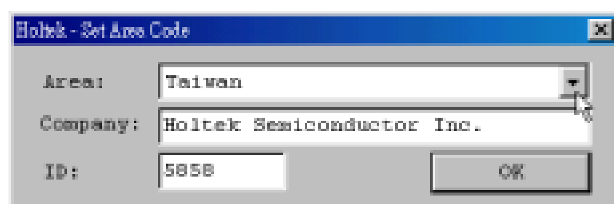


图 1-10

第二章

快速开始

2

本章简述如何快速使用 HT-IDE3000 去开发一个应用程序项目。

步骤一:建立一个新项目

- 按下 Project 菜单并选择 New 命令
- 输入项目名称并从组合框选择此项目使用的单片机型号
- 按下 OK 按键则系统将会要求设定单片机的掩膜选项
- 设定所有掩膜选项并按下 SAVE 键

步骤二:将源程序文件加到项目中

- 使用 File/New 命令建立源程序文件
- 撰写完程序后存盘，如 TEST.ASM 文件名
- 按下 Project 菜单并选择 Edit 命令
- 进入 Edit Project 对话框以便将源程序文件加到项目中
- 选择一个源程序文件名称，如 TEST.ASM，按下 Add 按钮
- 当所有源程序文件都被加入项目后，按下 OK 按钮

步骤三:完成项目

- 按下 Project 菜单并选择 Build 命令
- 系统将会对项目中的所有源程序文件执行编译动作
  - 若程序中有错误，只要在错误信息行连按两次，则系统将会提示错误发生的位置并且打开此错误所在的源程序文件，便可直接修改程序及储存文件
  - 若所有程序文件都没有错误，则系统会生成一个执行文件并且下载到 HT-ICE 中，准备仿真及除错
- 你可以重复上述步骤直到没有错误

步骤四:烧录 OTP 单片机

- 建立项目，并生成 .OTP 文件
- 按下 Tools 菜单并选择 HandyWriter 命令或者使用 HT-IDE3000 程序集中的 HT-HandyWriter 执行文件去烧录 OTP 芯片

步骤五:传送程序与掩膜选项单至 Holtek

- 按下 Project 菜单并选择 Print Option Table 命令去打印掩膜选项确认单
- 传送.COD 文件和掩膜选项确认单到盛群半导体公司，进行生产

程序及数据流程可由下图表示：

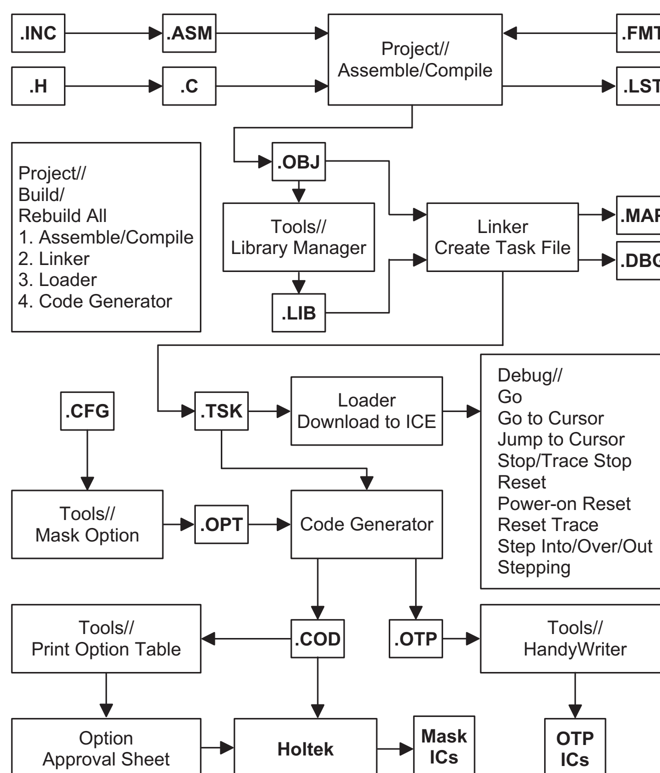


图 2-1

第三章

菜单 –
文件/编辑/视图/工具/选项

3

本章描述 HT-IDE3000 的部分菜单和命令，其它的菜单则在项目、除错及窗口等章节中叙述。

启动 HT-IDE3000 系统

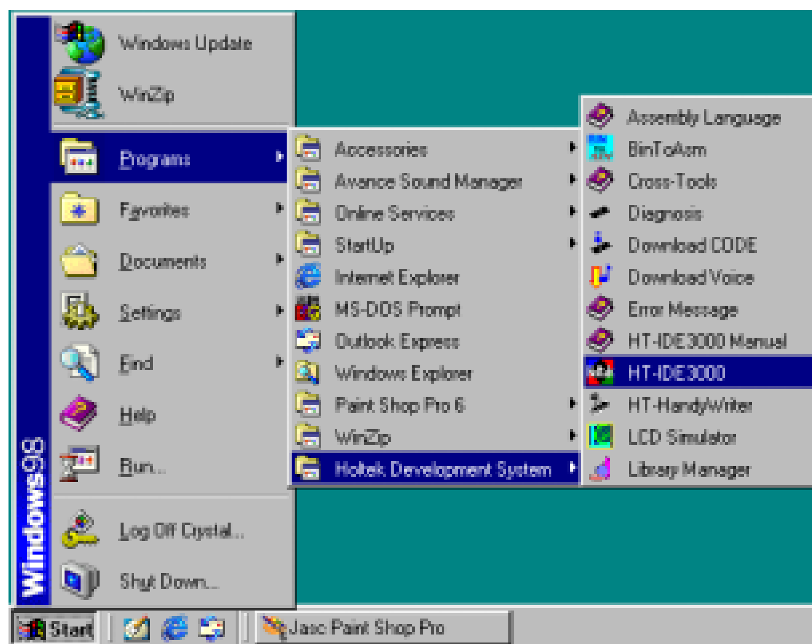


图 3-1

- 按下[开始]按钮，选择[程序]并选择 Holtek HT-IDE3000
  – 按下 HT-IDE3000 的小图标
- 若上次在 HT-IDE3000 中的项目是工作在硬件仿真模式（使用 HT-ICE)，如果下面的条件发生时，将会显示图 3-2 的画面。
  – 计算机与 HT-ICE 之间没有连接或连接失败。
  – HT-ICE 电源关闭。

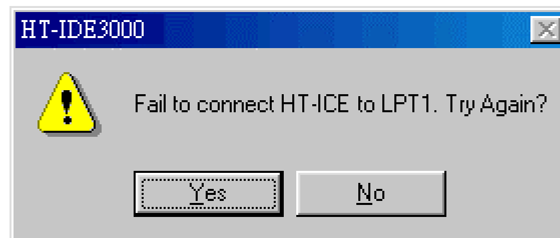


图 3-2

如果选择“ YES ”并且计算机与 HT-ICE 已经连接，则显示如图 3-3 的画面，HT-IDE3000 进入仿真模式，HT-ICE 开始动作。



图 3-3

- 若上次在 HT-IDE3000 中的项目是工作在软件仿真模式(使用软件仿真器)，则会显示如图 3-4 的画面，表示 HT-IDE3000 进入软件仿真模式。



图 3-4

HT-IDE3000 提供九种菜单-文件(File)、编辑(Edit)、视图(View)、项目(Project)、除错(debug)、工具(Tools)、选项(Options)、窗口(Windows)以及帮助(Help)，下面的章节将会分别描述每一个菜单的命令及功能。

在菜单栏下方的是工具栏（如图 3-5 所示)，可以随处移动此工具栏的位置。其中所包含的小图标，都是经常使用的菜单命令，可让使用者更方便地去执行菜单命令。将光标放置在工具栏的任何一个小图标上时，相对应的命令名称会显示在旁边，按下小图标就会执行此命令。

状态栏位于底线处（如图 3-5 所示)，显示硬件仿真或软件仿真的当前状态以及命令的执行结果状态。

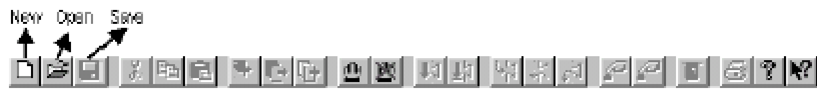
当执行除错时（除错菜单)，状态栏中的字段（PC=0001H）会显示程序计数器的内容。



图 3-5

状态栏所显示的信息可能在程序除错阶段有所帮助。例如程序计数器在程序执行期间会显示实际程序计数器的内容，而在使用程序编辑器时，行与列的指示器(状态栏最右边的两个字段)会显示光标当前的位置。

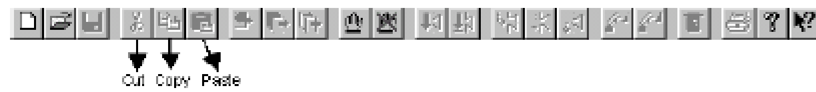
文件菜单



文件菜单提供文件处理命令，对应的工具栏小图标显示如上，细节说明如下：

- New
  建立新文件
- Open
  打开已有文件
- Close
  关闭当前文件
- Save
  储存当前文件
- Save As
  将当前文件另存成新文件名
- Save All
  储存所有被打开的文件
- Print
  将当前数据送到打印机进行打印
- Print Setup
  设定打印机
- Recent Files
  列出最近打开和关闭的四个文件
- Exit
  退出 HT-IDE3000，回到 Windows 窗口

编辑菜单



- Undo
  恢复前次的编辑动作
- Redo
  取消恢复的编辑动作
- Cut
  将选定的行从文件中移除并且把它存放到剪贴板（clipboard）
- Copy
  将选定的行复制到剪贴板
- Paste
  将剪贴板中的数据贴到当前的嵌入点
- Delete
  删除选定的数据
- Find
  在编辑窗口中寻找特定字符
- Replace
  将编辑窗口中指定的字符用特定字符取代

视图菜单

视图菜单提供下列命令控制 HT-IDE3000 的窗口（参考图 3-6）

- Line
  移动光标到文件的指定行处（由行号指定）
- Cycle Count
  累计指令的执行周期数。按下复位钮可清除计数周期，使用基数钮可改变计数的基数为十六进制或十进制，最大计数周期为 65535。
- Toolbar
  此命令可将工具栏的信息显示在窗口上或从窗口上取消，其中包含一些按钮群组，它的命令功能与每个对应的菜单项目相同。当鼠标光标放置在工具栏按钮上时，相对应的功能名称将显示在按钮之旁，若按下鼠标按钮则开始执行此命令，每一个按钮的功能请参考相对应的章节。Toggle Breakpoint 按钮可将光标所在位置的行设为断点（高亮标示），再次按这个钮则取消已经设定的断点。
- Status Bar
  打勾表示在窗口上显示状态栏信息。



图 3-6

工具菜单

工具菜单提供特别的命令帮助使用者程序除错，这些命令为 Mask Option、Diagnose、HandyWriter、Library Manager、Voice tools、LCD Simulator 及 virtual peripheral manager。

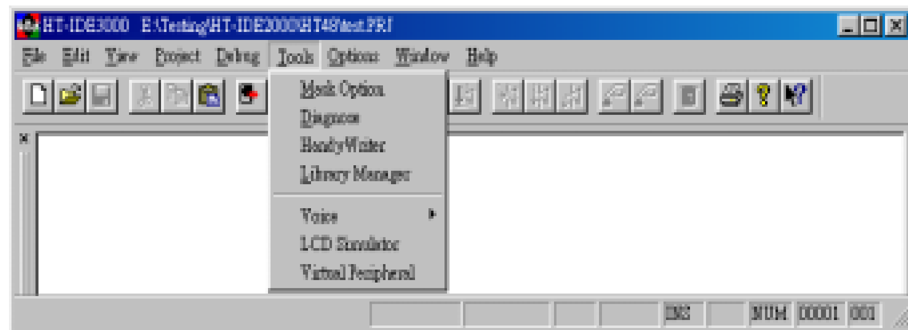


图 3-7

Mask Option

在建立一个新项目时(请参考第四章)，系统会根据使用者所选定的单片机以及外围配置，生成一个掩膜选项文件。在使用 Project 菜单的 Build 命令编译此项目的过程中，系统需要此掩膜选项文件以便生成执行文件。在项目建立后，可以使用 Mask Option 命令去修改单片机的掩膜选项，之后必须重新使用 Build 命令编译项目，否则修改无效。

→ **选择时钟源**

当建立一个新项目或是修改掩膜选项文件时，必须要设定 HT-ICE 所使用的时钟源。HT-ICE 提供内部及外部两种时钟源，若是选用外部的时钟源则 I/O 接口卡上的跳针 JP1 必须被跨接在正确的位置。

- 若使用晶体振荡模式，则需要将晶体振荡器插到 I/O 接口卡的 X1 位置，同时将 I/O 接口卡上跳针 JP1 的位置 2 与 3 短路。
- 若使用 RC 模式，则通过 VR1 调整系统频率，同时将 I/O 接口卡上跳针 JP1 的位置 1、2 短路。

→ **内部时钟源**

如果使用内部的时钟源，则必须指定 HT-ICE 的系统频率。HT-IDE3000 软件系统将会估算一个 HT-ICE 能够支持而且最近似于所指定的系统频率值。当估算频率不等于指定的频率时，将会显示警告信息以及指定的频率值与估算的频率值。客户必须确认要使用估算的频率或指定另一个系统频率，否则就只有使用外部的时钟源。无论选用何种时钟源，必须要指定系统频率。

Diagnose

诊断命令将验证 9 个项目，目的是帮助使用者检查 HT-ICE 是否可以正常工作（图 3-8）。可以选择验证数个项目然后按下 Test 键，也可按下 Test All 键去诊断所有项目。这些项目如下所列。

- MCU resource option space
  诊断 HT-ICE 的单片机掩膜选项空间
- Code space
  诊断 HT-ICE 的程序代码存储器
- Trace space
  诊断 HT-ICE 的跟踪存储器
- Data space
  诊断 HT-ICE 的程序数据存储器
- System space
  诊断 HT-ICE 的系统数据存储器
- I/O EV 0
  诊断 HT-ICE 在插槽 0 的 I/O 仿真模块
- I/O EV 1
  诊断 HT-ICE 在插槽 1 的 I/O 仿真模块
- I/O EV 2
  诊断 HT-ICE 在插槽 2 的 I/O 仿真模块
- I/O EV 3
  诊断 HT-ICE 在插槽 3 的 I/O 仿真模块

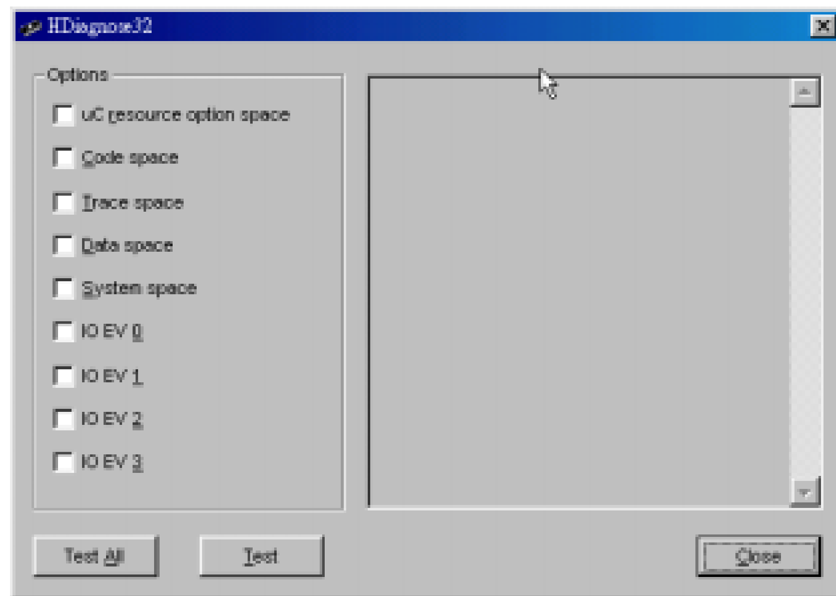


图 3-8

HandyWriter

HandyWriter 提供烧录 OTP 芯片的功能，请参考 OTP 烧录的章节。此命令的功能与 HT-IDE3000 群组中的 HandyWriter 小图标相同。

Library Manager

在图 3-9 中 Library Manager 命令支持函数库功能。经常使用的程序代码可以编译成函数文件，然后通过 Option 菜单中的 Project 命令把它引用到应用程序内（请参考 Options 菜单中的 Project 命令的 Cross Linker 选项）。Library Manager 的功能如下：

- 建立新的函数库文件或修改函数库文件。
- 将程序模块加到函数库文件或从函数库文件删除程序模块。
- 从函数库文件中取出程序模块然后建立一个目标文件。

有关 Library Manager 更详细的资料，会在第三部份说明。

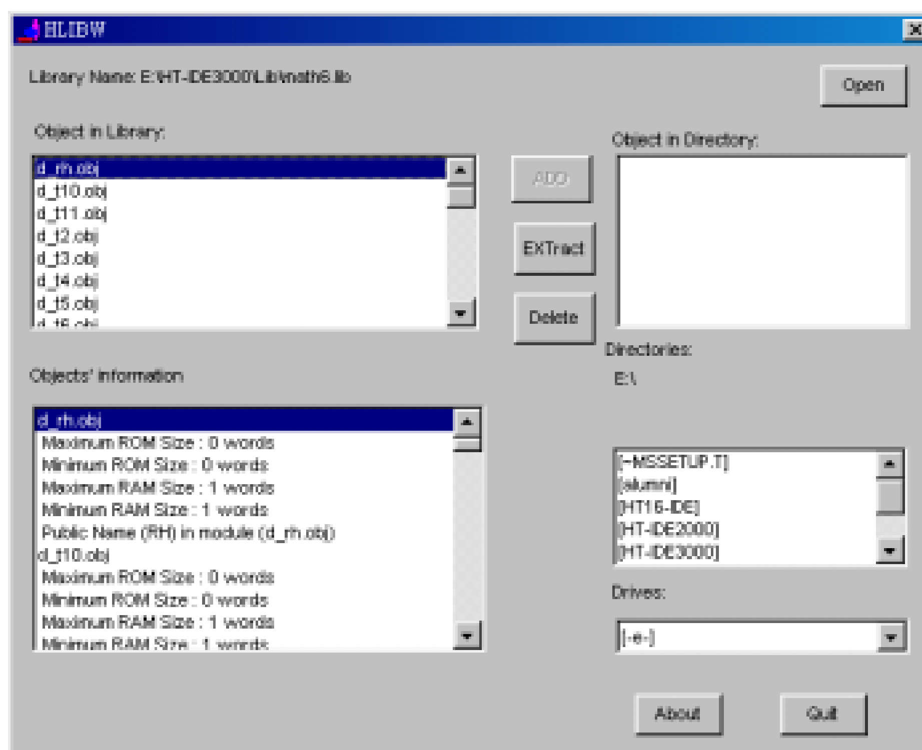


图 3-9

Voice/VROM Editor

对于特别指定的单片机（例如 HT86 系列），此命令提供 VROM 编辑器，让使用者可安排语音码。

Voice/Download

此命令下载后缀名为.VOC 的语音文件内容到 HT-ICE 做硬件仿真，也可以从 HT-ICE VROM 上传储存数据到指定的.VOC 文档。图 3-10 的对话框显示由 VROM 编辑器所生成的下载语音文件.VOC 的文件名，文件名下方的方框会显示项目中单片机的语音数据所需要的存储器大小（单位 Bytes）。当上传时，可以指定不同于项目名称的文件名来储存 HT-ICE 中语音存储器的内容。在下载语音文件之前，必须确定语音 ROM 文件.VOC 已通过 VROM 编辑器生成。

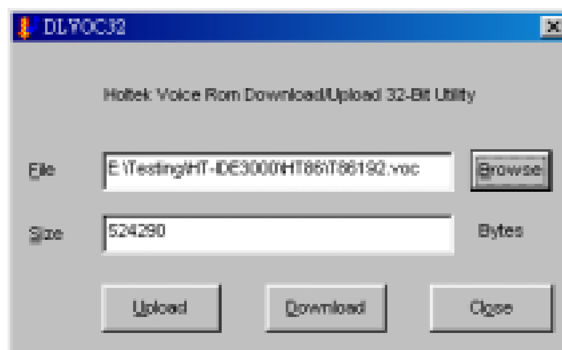


图 3-10

LCD Simulator

LCD 软件仿真器，简称 HT-LCDS，提供 LCD 驱动的软件仿真。根据设计好的图形和控制程序，HT-LCDS 能够实时将图形显示在显示器上。有关 LCD simulator 更详细的资料，会在第三部份加以说明。

Virtual Peripheral Manager

Virtual Peripheral Manager（VPM）提供外围器件的软件仿真，它必须在 HT-IDE3000 处于软件仿真模式下才能使用。

选项菜单

选项菜单（图 3-11）提供下列命令，可对其它菜单与命令设定工作参数。

Project Command

Project 命令设定 Project 菜单中 Build 命令的默认参数值。在程序开发期间，项目选项依据应用程序的需求而改变。HT-IDE3000 在执行 Project 菜单中的 Build 命令时，将根据这些选项的设定生成对应的任务文件及内容。对话框（图 3-12）用来设定 Project 的选项。

注意： 在执行 Build 命令之前，要先确认项目选项的设定正确。

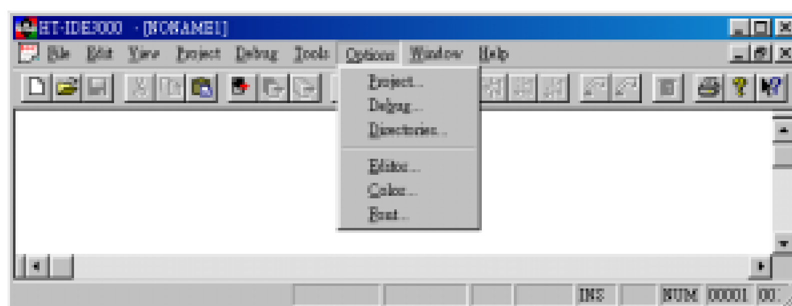


图 3-11

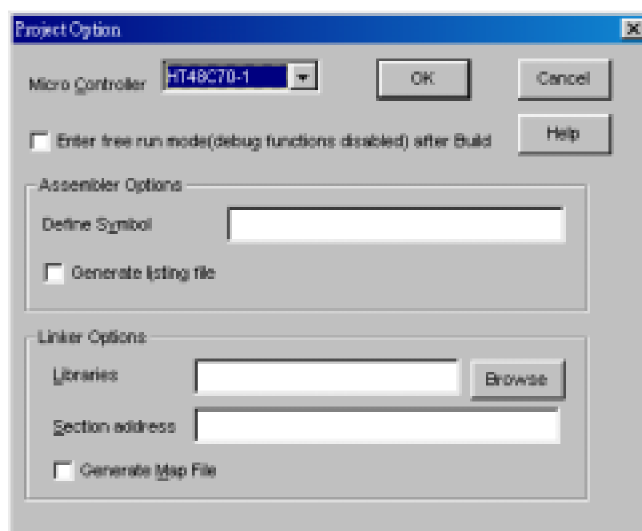


图 3-12

- Micro Controller
  项目中的单片机名称。使用滚动条浏览可使用的单片机，并选择合适的单片机。
- Enter free run mode（debug options disabled）after build
  选中此对话框将会命令 HT-IDE3000 编译出自动执行模式的项目执行文件。自动执行模式的项目执行文件只能执行程序，所有除错功能都无法运作。
- Cross Assembler options
  编译器的命令行选项。Define Symbol 栏允许使用者对指定的符号定义数值，而所指定的符号则在编译过程中使用。语法如下：

  ```
  symbol1[=value1] [,symbol2 [=value2] [,...]]
  ```

  例如：

  ```
  debugflag=1,  newver=3
  ```

  另外，可选中生成列表文件(Generate listing file)检查框以便生成源程序列表文件。
- Cross Linker options
  指定连接器的选项。Libraries 文字框用来指定被连接器所参考使用的函数库文件。例如：

  ```
  libfile1,  libfile2
  ```

  也可以按下 Browse 键来浏览及选择函数库文件。
  另外，程序段地址(Section address)输入框则是用来设定程序段的 ROM/RAM 地址。例如：

  ```
  codesec=100,  datasec=40
  ```

  最后一行的映射文件检查框(Generate map file)则是命令连接器生成项目的程序映射文件。

Debug Command

此命令设定供 Debug 菜单使用的选项功能（第五章 HT-IDE3000 菜单-除错）。对话框（图 3-13）列出附带有检查框的除错选项，当完成选项的选择并按下 OK 钮之后，除错过程中 Debug 菜单及命令就根据这些选项执行对应的动作。

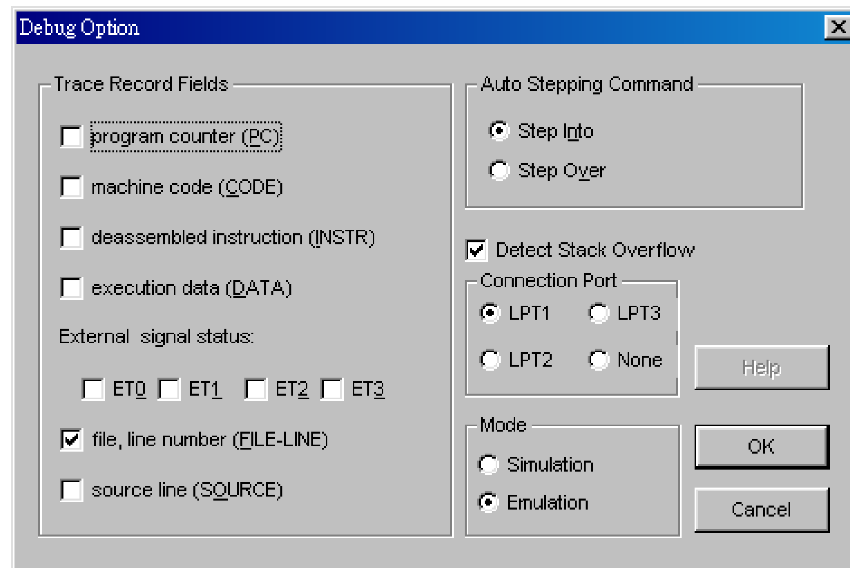


图 3-13

- Trace Record Fields

  此字段中的各选项将会影响程序跟踪(trace)时所记录的数据是否要显示在窗口内。在执行 Window 菜单的 Trace List 命令时，这些选项会指示哪些跟踪数据需要显示出来。对于每个源程序文件内的指令，信息的显示顺序将和对话框项目的排列秩序一致，从上往下。如果其中的项目没有被选取，则下一个被选取的项目的数据将往前移。跟踪列表的默认数据只会显示文件名称和各行的行号。program counter 是指令的存储器地址，machine code 是指令的机器代码，de-assembled instruction 是从机器代码直接反译回来的指令，而 source line 是源程序文件的指令行。

  如果是只读动作的指令，execution data 显示的是读取的数据，然而若是只写或读写动作的指令，则显示被写入的数据。file，line number 是要显示源程序文件名与指令行的行号。external signal status 则是选择是否要显示外部输入的信号。当选择软件仿真模式时，external signal status 无效。

- Auto Stepping Command

  此选项可以选择自动调用程序的执行步骤，即 Step Into 或 Step Over。只能选择其中之一。

- Connection Port
  选择 HT-ICE 连接口。个人计算机的任何一个并行端口 LPT1，LPT2 或 LPT3 都可被选取与 HT-ICE 连接。如果选择软件仿真模式，则连接口无用。
- Mode
  选择 HT-IDE3000 的工作模式为软件仿真或硬件仿真模式。如果 HT-ICE 被连接到个人计算机且电源已打开，则 HT-IDE3000 可以选择软件仿真或是硬件仿真模式。
- Detect Stack Overflow
  如果你不需要系统在发现堆栈溢出时显示警告信息，请不要勾选此对话框。

Directories Command

这个命令用来设定查找执行文件或储存输出文件的默认路径及文件夹（图 3-14）。

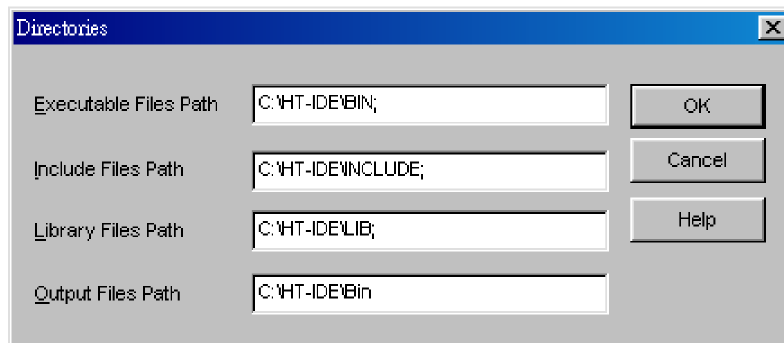


图 3-14

- Executable files path
  HT-IDE3000 会从此处所设定的路径或文件夹去查找将要被执行的文件。
- Include files path
  编译器会从此处设定的路径或文件夹查找包含文件。
- Library files path
  连接器会从此处设定的路径或文件夹查找函数库文件。
- Output files path
  储存编译器的输出文件（.obj，.list）和连接器的输出文件（.tsk，.map，.dbg）所使用的路径。

Editor Command

这个命令会设定编辑器选项，如 TAB 键的大小和 Undo 的有效次数。Save Before Assemble 选项是要在编译前预先储存文件。Maximum Undo Count 是可以连续执行恢复动作的最大次数。

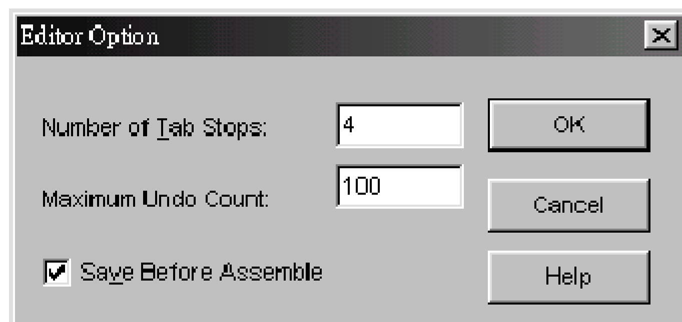


图 3-15

Color Command

这个命令可以设定指定行的前景和背景颜色。在图 3-16 左边的选项中，Text Selection 用在 Edit 菜单，Current Line，Breakpoint Line，Trace Line 和 Stack Line 用在 Debug 菜单，而 Error Line 则是针对编译器的输出。

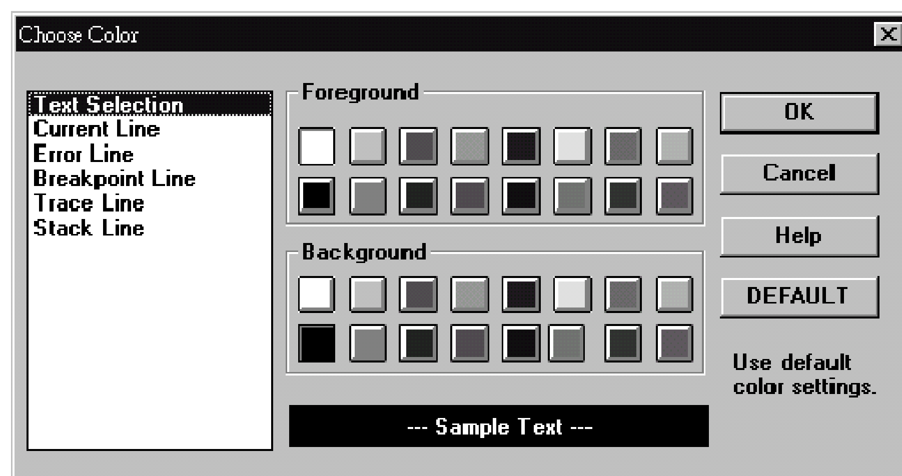


图 3-16

Font Command

这个命令可以改变显示的字型。

第四章

菜单 – 项目

4

HT-IDE3000 提供项目范例帮助第一次使用者快速地熟悉项目开发流程，从 HT-IDE3000 系统的观点来看，一个工作单元是由包含应用程序的项目所描述。

当首次开发一个 HT-IDE3000 的应用程序时，应遵循先前所建议的开发步骤进行。

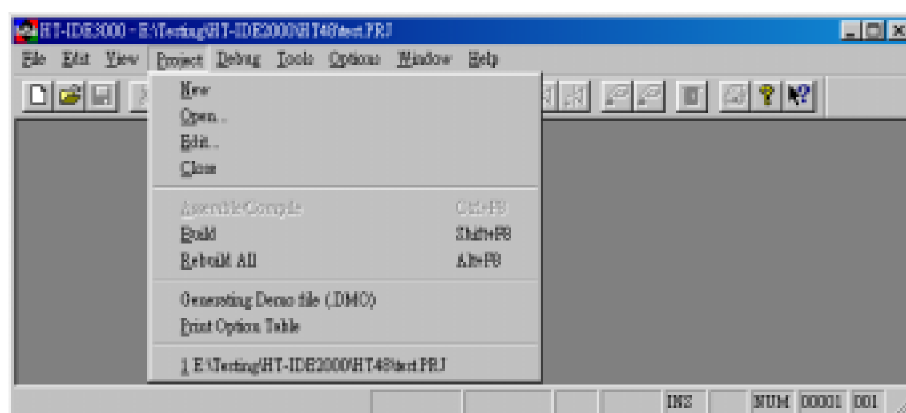


图 4-1

建立新项目



在 Project 菜单（图 4-1）中选择 New 命令，建立一个新项目，此命令会要求使用者输入或选择两项信息，即 Project Name 和 Micro Controller（图 4-2）。使用者可以按下浏览（Browse）键，选取期望的路径，新建项目或覆盖原有项目。另外再选择此项目使用的单片机型号。

注意： 项目名称的后缀名为.PRJ

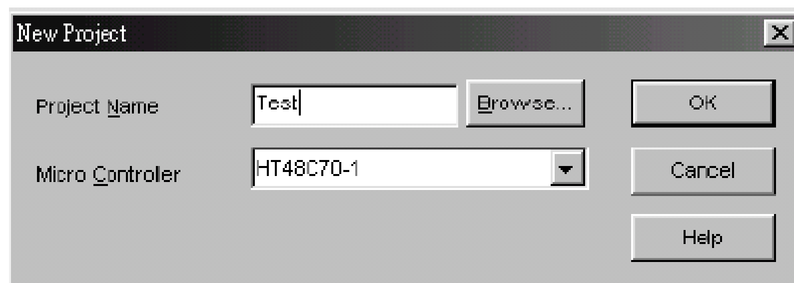


图 4-2

打开和关闭项目

任何时候 HT-IDE3000 只能执行一个项目，就是已经打开的项目。如果要使项目开始工作，则首先要使用 Project 菜单的 Open 命令打开此项目（图 4-1），可直接输入项目名称，或浏览文件夹选择项目名称。Close 命令将关闭当前使用的项目。

注意： 当打开一个项目时，则当前已打开的项目会自动地关闭。

在开发过程中，亦即在编辑、设定选项及除错的阶段时，必须确定项目在打开的状态。可以从显示在 HT-IDE3000 窗口上方的打开项目名称得知。如果项目不在打开状态下，则结果是不可预测的。

当结束 HT-IDE3000 的执行回到一般窗口时，如果没有关闭已打开的项目，则系统会将此项目的数据保存起来，在下次执行 HT-IDE3000 时，系统会自动去打开此项目。

管理项目的源文件

项目中必须要有源程序文件，使用 Project 菜单的 Edit 命令可以将源程序文件加到项目中，也可从项目中将源程序文件删除。列表框中源程序文件由上而下的顺序就是送入连接器的文件的顺序，连接器会根据列表框中文件的顺序去处理这些输入文件。可以使用[Move Up]与[Move Down]按钮重新调整文件的排列顺序。图 4-3 是 Project 菜单中 Edit 命令的对话框。

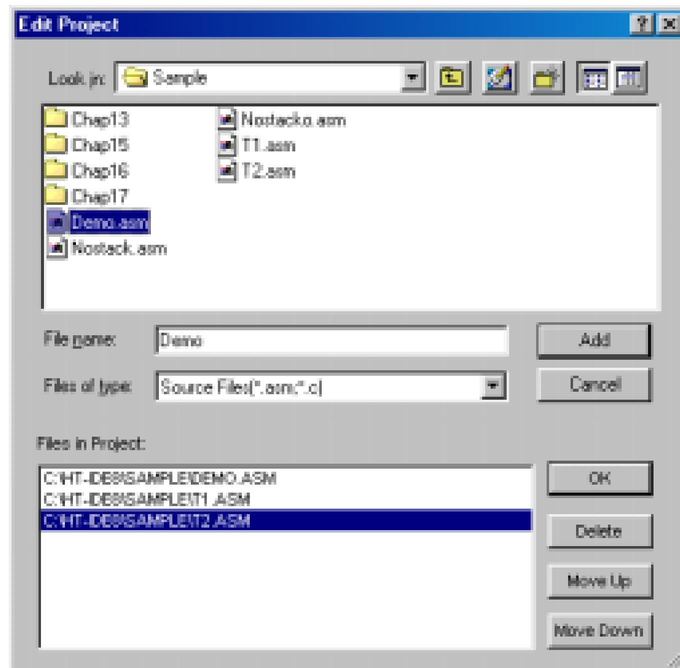


图 4-3

将源程序文件加到项目中

- 在 Edit Project 对话框中的 File name 文字框中输入源程序文件的文件名。
- 另外一种方法是选择源文件的文件类型 Files of type，再去浏览文件列表。
  - 通过浏览 Drives 和 Directories 的方式(从 Look in 浏览框中选取)，选定源程序文件所在的驱动器和路径。
  - 从文件名称(File name)项目上方的列表框中选取源程序文件名称。
  - 双击所选取的文件名称或是按下[Add]钮，将源程序文件加到项目中。

当所选取的源程序文件被加入到项目后，文件的名称会显示在项目中的文件列表框上(Files in Project)。

从项目中删除源程序文件

- 从 Files in Project 列表框中选择欲删除的文件。
- 按下 Delete 钮。

注意: 从项目中删除源程序文件并不会真正将文件删除，只是将文件数据从项目中移除。

向上或向下移动源程序文件的位置

- 在 Files in Project 列表框中，将光标移到指定的文件处，并按下鼠标左钮。
- 再按下[Move Up]钮或[Move Down]钮，将指定的文件往上移或往下移动位置。

建立项目的任务文件

编译新的项目之前，应确定下列工作已完成：

- 项目已经打开
- 已完成项目选项的设定
- 已经将源程序文件加到项目中
- 已选定项目所使用的单片机（参考 Tools 菜单章节）

有两个编译项目文件的命令，分别为 Build 命令和 Rebuild All 命令。

Build 命令会执行下列的工作：

- 根据程序文件的后缀名.asm 或.c 分别调用汇编编译器或 C 编译器去编译项目中所有的源程序文件
- 将汇编编译器或 C 编译器生成的目标文件连接，并生成任务文件和除错文件
- 如果 HT-ICE 的电源已打开，并且与个人计算机相连，则下载任务文件至 HT-ICE
- 将开始执行点的源程序显示在工作窗口上（HT-IDE3000 硬件仿真须参考源文件、任务文件和除错文件）

注意: Build 命令会根据相关文件建立的日期与时间决定上述那些步骤会被执行或不被执行。规则如下：

- 若源程序文件的生成日期/时间晚于它的目标文件的生成日期及时间，则 Build 命令会调用汇编编译器或 C 编译器去编译源程序文件并生成新的目标文件。
- 若有一个目标文件的生成日期/时间晚于任务文件的生成时间，则 Build 命令会调用连接器将所有目标文件连接起来并生成新的任务文件。

无论上述的步骤是否都被执行，Build 命令会自动将任务文件的内容下载到HT-ICE。

Rebuild All 命令与 Build 命令执行的工作相同，其差异在于 Rebuild All 会立即执行工作，而不会先比较所有文件的生成日期/时间。

Build 或 Rebuild 命令执行结果的信息会显示在 Output 窗口。在处理过程中如果发生错误，其后的步骤会被忽略，不会生成任务文件及下载任务文件。

建立项目的任务文件

- 按下 Project 菜单中的 Open 命令，打开项目。
- 按下 Project 菜单中的 Build 命令或工具栏中的 Build 键（图 4-1），开始编译这个项目。

重建项目的任务文件

- 按下 Project 菜单中的 Open 命令，打开项目。
- 按下 Project 菜单中的 Rebuild All 命令或工具栏中的 Rebuild All 键(图 4-1)，开始编译这个项目。

一旦编译好项目的任务文件，就可以开始进行应用程序的仿真和除错（参考 HT-IDE3000 菜单-除错章节）。

编译

可以先用此命令去编译源程序以及在 Output 窗口上显示其结果的方式，来验证应用程序的正确与否。因为一次只会编译一个源程序文件，使用者可以不用一次面对整个项目中所有源程序文件编译的结果，尤其是不只一个程序文件发生错误时。

编译程序

- 使用 File 菜单 Open 命令打开源程序文件以进行编译
- 选用 Project 菜单的 Assemble/Compile 命令或按下工具栏的 Assemble 键去编译程序文件

假如已打开文件的后缀名为.asm，则系统会调用汇编编译器执行编译工作。如果已打开文件的后缀名为.c，则系统会调用 C 编译器去编译程序。

如果没有发生错误，则会生成后缀名为.OBJ 的目标文件，并且储存在 Output Files Path 所指定的目录内（参考 Options 菜单中的 Directories 命令）。如果发生错误，则在 Output 窗口上显示错误信息，可使用下列的方式将错误行所在的源程序文件打开并使光标停留在错误行。

- 双击鼠标左键或
- 将光标移到 Output 窗口内的错误信息行，按下<Enter>键

Print Option Table Command

此命令会将当前使用中的掩膜选项文件的内容送到指定的打印机上进行打印，可以选用不同的打印机去打印。建议不要使用连接至 HT-ICE 的打印机端口。

假如打印机和 HT-ICE 使用相同的打印机端口，则发出此命令时会造成除错信息与相关数据的丢失。在打印工作结束后，假如应用程序仍需做硬件仿真与除错，则使用者应回到开发过程的开始处，再使用 Project 菜单中的 Build 指令做下载动作。

Generate Demo File (.DMO) Command

此命令将会生成一个可供 HT-DEMO 使用的.dmo 文件，使用者只需携带 HT-DEMO 和.dmo 文件，在未安装或未执行 HT-IDE3000 的计算机上执行项目的程序。

第五章

菜单 – 除错

5

在程序开发的过程中，重复的修正和测试源程序是不可避免的。HT-IDE3000 提供许多工具，不仅让除错工作更加容易，而且也减少开发时间。这些功能包含了单步执行、符号断点、自动单步执行、跟踪触发条件等。

当应用程序成功建立之后（参考前一章中建立项目的任务文件），在工作窗口（图 5-1）中的源程序代码，被高亮标示的程序行，将是第一个被执行的程序行。之后，系统准备接受及执行使用者所下的除错命令。

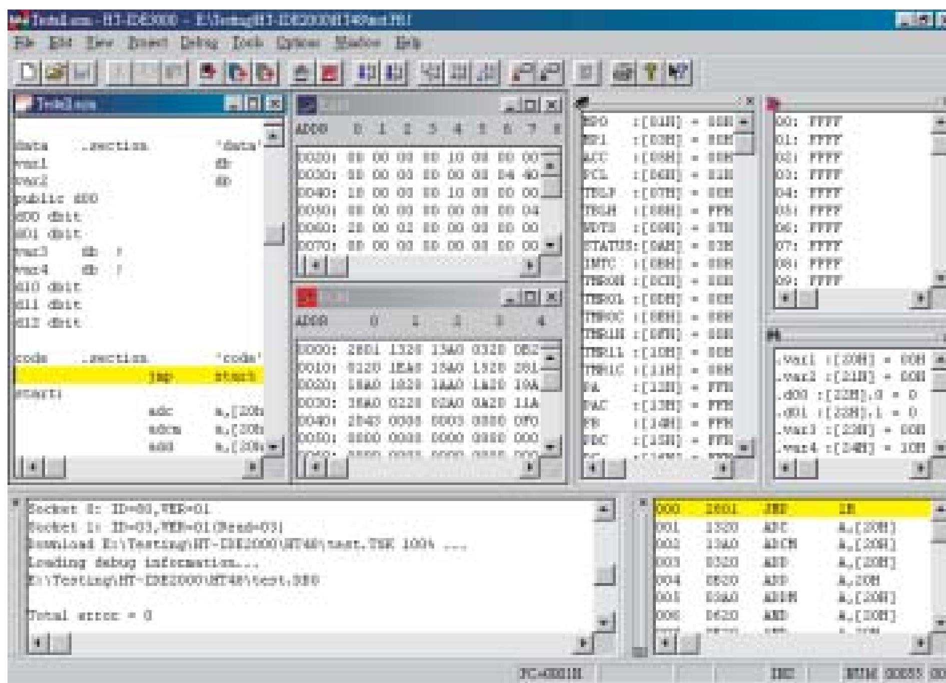


图 5-1

复位 HT-IDE3000 系统

HT-IDE3000 系统提供四种复位方法：

- 电源复位（Power-on reset，POR）通过将电源变压器接到 HT-ICE 或按下 HT-ICE 上的复位钮。
- 来自应用电路板的复位信号。
- 来自 HT-IDE3000 中 Debug 菜单（图 5-2）的软件复位 Reset 命令。
- 来自 HT-IDE3000 中 Debug 菜单（图 5-2）的软件电源复位 Power-on Reset 命令。

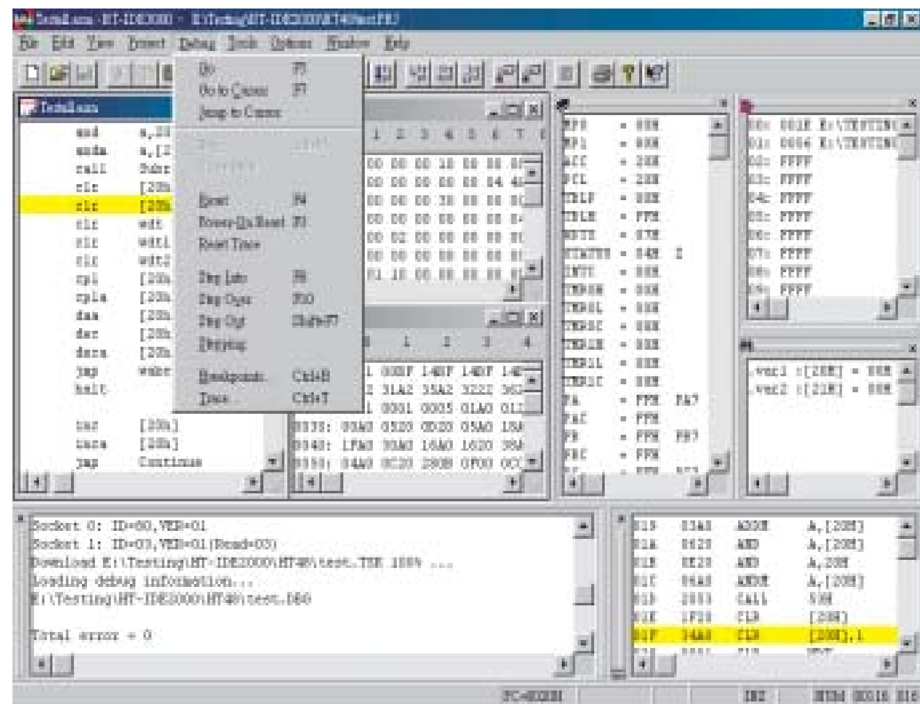


图 5-2

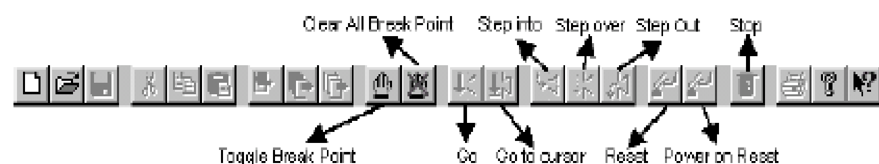


表 5-1 列出上述四种复位的影响。

Reset Item	Power-On Reset	Target Board Reset	Software Reset Command	Software Power-On Reset Command
Clear Registers	(*)	(*)	(*)	(*)
Clear Options	Yes	No	No	No
Clear PD，TO	Yes	No	No	Yes
PC Value	(**)	0	0	0
Emulation Stop	(**)	No(***)	Yes	Yes
Check Stand-Alone	Yes	No	No	No

表 5-1

注意: (*)：寄存器在不同复位下所受的影响，请参考各单片机的规格书。

(**)：PC 值为 0，同时停止硬件仿真。

(***)：假如复位来自于应用电路板，则系统会在复位完成后自动开始硬件的仿真。

PC-程序计数器

PD-暂停标志位

TO-WDT 溢出标志位

从 HT-IDE3000 复位

- 选按 Debug 菜单的 Reset 命令或按下工具栏的 Reset 钮去执行软件复位。
- 选按 Debug 菜单的 Power-on Reset 命令或按下工具栏的 Power-On Reset 钮去执行复位。

从应用电路板复位

利用 DIN 连接器上的μ_RES 引脚（pin 03-C）在应用电路板上设计单片机的复位钮，这个复位后的结果列在表 5-1 中。

应用程序的硬件仿真

在应用程序撰写完成后，需要使用 Build 或 Rebuild 命令建立项目的工作文件以及将程序下载到 HT-ICE。如果此命令执行完毕，没有发生错误，则在工作窗口（图 5-1）中高亮标示源程序中最先被执行的程序行。此时，可以使用 HT-IDE3000 的除错命令开始进行应用程序的仿真。

注意：在应用程序的硬件仿真期间，对应的项目必须已经被打开。

硬件仿真应用程序

- 选用 Debug 菜单的 Go 命令
  或按下快捷键 F5
  或按下工具栏的 Go 按钮

在硬件仿真期间可以启动其它的窗口。如果断点条件符合的话，HT-IDE3000 系统将会自动停止仿真，否则它会继续执行直到应用程序结束为止。当 HT-ICE 在硬件仿真时，工具栏的 Stop 钮会呈现红色，代表此时 Stop 钮是有效的。如按下 Stop 钮则仿真会立刻停止。Stop 钮在仿真停止状态下会呈现灰色，代表此时 Stop 钮是无效的。

停止硬件仿真应用程序

共有三种停止硬件仿真的方法，如下所示：

- 开始硬件仿真之前先设定断点。
- 选用 Debug 菜单的 Stop 命令或按下快捷键 Alt+F5。
- 按下工具栏的 Stop 钮。

执行应用程序到指定的程序行

在进行程序的除错过程中，可以命令仿真动作停在指定的程序行。下列的方法提供此种功能。除了条件跳行指令之外，从当前停在的程序行到指定程序行之间所有的指令都会被执行。由于条件跳行或其它情况，程序执行的结果也许不会停留在指定的程序行。

- 移动光标到停止的程序行（高亮标示此行）。
- 选择 Debug 菜单的 Go to Cursor 命令
  或按下快捷键 F7
  或按下工具栏的 Go to Cursor 钮

直接跳跃到应用程序中的某一行

如果从当前停留的程序行到指定程序行之间所有的指令，其执行结果不影响除错效能，则可以直接跳跃至某一程序行。除了程序计数器之外，此命令不会改变数据存储器、寄存器和状态寄存器的内容，指定的程序行就是下一条要被执行的指令。

- 移动光标至要跳跃到的程序行或高亮标示此行。
- 选用 Debug 菜单的 Jump to Cursor 命令。

单步执行

以上章节所描述的除错命令是查看与侦错数个程序行执行的结果，如果想要执行一行程序之后就去查看其结果，则需要使用单步执行命令。HT-IDE3000 提供两种单步执行模式，手动模式与自动模式。

手动模式下，使用者每发出一个单步执行命令，HT-IDE3000 只会执行一次单步执行命令。但是在自动模式下，HT-IDE3000 则会重复执行单步执行命令直到用 Debug 菜单的 Stop 命令让硬件仿真停止。在自动模式下，所有设定的断点都无效，而单步执行的速率可以被设定为 FAST、0.5、1、2、3、4 和 5 秒。总共有三个单步执行命令，分别为 Step Into、Step Over 和 Step Out。

- Step Into 命令一次只执行一条指令就停止，但是如果所执行的指令是 CALL 程序指令时，则会进入此子程序并且停在子程序中的第一条指令处。
- Step Over 命令一次只执行一条指令就停止，但是如果所执行的指令是 CALL 程序指令时，则不会进入子程序，而是停留在 CALL 指令后的下一条指令处，此子程序中所有的指令会被执行，而且寄存器与状态寄存器的内容也根据执行的结果做改变。
- Step out 命令只能在仿真停在子程序之内时被使用，它会执行当前停在的程序行和 RET 指令（包括 RET 指令）之间所有的指令，然后停留在 CALL 指令后的下一条指令处。

注意： Step out 命令只能被使用在当前的停止点在子程序内的情况下，否则会发生无法预期的结果。

如果要使用自动模式，则需要先使用 Options 菜单的 Debug 命令去设定自动模式的单步执行命令，即 Step Into 或是 Step Over。

- 启动单步执行的自动模式
  从 Debug 菜单中选择 Stepping 命令，也选择单步执行的速度 Stepping Speed（使用 Options 菜单的 Debug 命令去选择单步执行命令: Step Into 或 Step Over）
- 结束单步执行的自动模式
  从 Debug 菜单中选择 Stop 命令
- 改变自动模式下的单步执行命令
  – 从 Options 菜单中选择 Debug 命令
  – 在 Stepping 命令框中选择 Step Into 或 Step Over 命令
- 启动 Step Into 命令
  从 Debug 菜单中选择 Step Into 命令
  或按下快捷键 F8
  或按下工具栏的 Step Into 钮
- 启动 Step Over 命令
  从 Debug 菜单中选择 Step Over 命令
  或按下快捷键 F10
  或按下工具栏的 Step Over 钮
- 启动 Step Out 命令
  从 Debug 菜单中选择 Step Out 命令
  或按下快捷键 Shift + F7
  或按下工具栏的 Step Out 钮

断点

HT-IDE3000 提供一个有力的断点功能，可接受多种形式的条件，包括程序地址、源程序行号及符号中断等。

断点特性

HT-IDE3000 断点功能的主要特性如下：

- 任何时刻，最多有 3 个具有相同优先级的断点同时有效。
- 任何断点被设定之后，它会被记录在断点列表框（Breakpoints）内，然而此断点或许不会立即有效，但是只要它不被删除（即仍然在断点列表框内），稍后可将它再设为有效。
- 任何时刻，断点列表框中最多可加入 20 个断点。如果要加入第 21 个，则必须从原先的 20 个断点中至少删除一个。
- 允许使用二进制形式（含 don't care）的地址与数据断点。
- 当指令行被设定为有效的断点时，HT-ICE 会停在这个指令行但是不会执行此指令，即这条指令将是下条被执行的指令。虽然指令行被设定为有效的断点，但由于执行流程或条件跳转的缘故，HT-ICE 可能不会停留在这个指令行。如果有效的断点是在 Data Space（RAM），符合断点数据的指令将会被执行，且 HT-ICE 会停在下一个指令行。

断点项目的说明

下面说明断点的描述项目及内容，如图 5-3 所示，并非所有的项目都需要设定：

- Space
  断点的位置空间，可以设定为程序代码空间或数据空间。
- Location
  断点的确切位置。下一节会说明位置的格式。
- Content
  断点的数据内容。此项目只有当位置空间被设为数据空间时才有效，同时需要使用 Read 和 Write 检查框设定数据的存取类型作为中断的执行条件。
- Externals
  外部信号断点。共有四个外部信号，ET0、ET1、ET2 及 ET3，位置在 I/O 接口卡上的 JP3。

→ **描述项目的格式—位置**

位置项目的格式如下：

- 共有四种形式的绝对地址（在程序代码空间或数据空间），称为：十进制、十六进制（字尾为“ H ”或“ h ”）、二进制及全部符合的二进制位(don’t-care bits)，例如：

  20, 14h , 00010100b , 10xx0011

  表示十进制的 20、十六进制的 14h、二进制的 00010100b 及都符合的位 4 和 5。

注意: Don’t-care bits 必须是二进制的形式。

- 具有或没有源文件名称的行号，格式如下：

  `[source_file_name!].line_number`

  这里的 **`source_file_name`** 是源文件的名称，可有或没有。如果没有文件名，则会以当前工作中的文件为准。如果指定了源文件名，则必须在文件名之后加上惊叹号“ !”，而句点“ . ”必须放在行号之前，格式为十进制。例如：

  `C:\HIDE\USER\GE.ASM!.42`

  将断点设定在 C 盘文件夹\HIDE\USER 内，文件名为 GE.ASM 的第 42 行。例如：

  `.48`

  将断点设在当前工作中的文件的第 48 行。

- 具有或没有源文件名称的程序符号，格式如下：

  `[source_file_name!].symbol_name`

  除了将 line_number 换为 symbol_name 外，其格式及内容与上述的行号位置格式相同。下列的程序符号都可被接受：

  – Label name 标号名

  – Section name 程序段的名称

  – Procedure name 子程序的名称

  – Dynamic data symbols defined in data section 数据段中所定义的数据变量

→ **描述项目的格式 – 内容和外部信号**

- 内容和外部信号的格式用四个数字表示，类似于位置项目的格式。这四种数字格式分别为十进制、十六进制（字尾为“ H ”或“ h ”）、二进制及全部符合的二进制位(don’t-care bits)。

→ **断点列表框的格式**

断点列表框包含所有已加入的断点，包括有效及无效的断点。Add 钮可用于增加新的断点到列表框，Delete 钮则是从列表框中移除断点。每个断点在列表框中的格式如下：

```
<status>  {<space and read/write>,  <location>,
<data content>,  <external signal>}
```

这里的<status>是有效状态，“ + ”表示有效（使能）而“ - ”表示无效（除能），<space and read/write>是空间形式及操作模式，“ C ”是程序代码存储器空间，“ D/R ”是从数据存储器空间读取，“ D/W ”是写入数据存储器空间，“ D/RW ”是读取与写入数据存储器空间。

<location>、<data content>及<external signal>分别与输入格式相同。

如何设定断点

有两种方法设置/使能断点，一种是使用 Debug 菜单的 Breakpoint 命令；另外一种是使用工具栏的 Toggle Breakpoint 钮。断点的规则如下：

- 如果断点没有被加到断点列表框（图 5-3），则要先设定各描述项目，然后再加入断点列表框。
- 只要断点存在于列表框中，即使先前此断点是无效的，也可以通过使能断点设为有效状态。
- 最后必须按下 OK 钮以确认设定，否则所有的更改都无效。
- 如果要用工具栏的 Toggle Breakpoint 钮去设定断点，则应先将光标移到断点所在的程序行，再按下 Toggle Breakpoint 钮，此时这个断点就被设为有效。如果要将有效的断点改变为无效的断点，则可再按下 Toggle Breakpoint 钮去改变。

→ **增加断点**

- 从 Debug 菜单中选择 Breakpoint 命令（或按下快捷键 Ctrl+B），断点对话框会出现在窗口中（图 5-3）。
- 设定断点的各描述项目
  设定 Space，Location 项目。
  如果 Space 是数据存储器空间，则需要设定 Content 项目和 Read/Write 检查框。
  如有需要，则要设定 External Signals。
- 按下 Add 钮将此断点加到断点列表框。
- 按下 OK 钮确认设定。

注意： 假如有效断点的总数小于 3 个，则新加入的断点将会自动生效。
假如断点列表框已有 20 个断点，则 Add 钮将会无效而不能再加入断点。

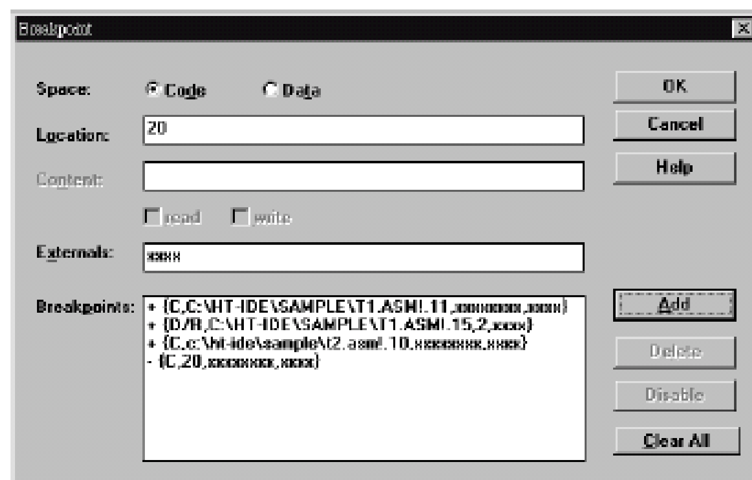


图 5-3

→ **删除一个断点**

- 从 Debug 菜单中选择 Breakpoint 命令或按下快捷键 Ctrl+B，断点对话框会出现在窗口中（图 5-3）。
- 从断点列表框中选择即高亮标示要被删除的断点。
- 按下 Delete 钮则会将此断点从断点列表框中删除。
- 按下 OK 钮确认设定。

→ **删除所有断点**

- 从 Debug 菜单中选择 Breakpoint 命令或按下快捷键 Ctrl+B，断点对话框会出现在窗口中（图 5-3）。
- 从断点列表框中选择 Clear All 钮以删除所有的断点。
- 按下 OK 钮确认设定。
- 也可以按下工具栏的 Clear All Breakpoint 钮去完成这项工作。

→ **设定一个断点为有效或无效**

- 从 Debug 菜单中选择 Breakpoint 命令或按下快捷键 Ctrl+B，断点对话框会出现在窗口中（图 5-3）。
- 从断点列表框中选择要设定的断点。
- 按下 Enable（Disable）钮去设定此断点为有效(无效)。
- 按下 OK 钮确认设定。

跟踪应用程序

当 HT-IDE3000 仿真应用程序时，它提供一个有力的跟踪工具，以记录执行过程与所有相关的信息。此工具还提供跟踪过滤器(qualifier)去过滤需要被跟踪的指令以及停止跟踪触发器(trigger)以便停止记录跟踪的数据。另外，它提供一种方法，可以指定在跟踪触发点(trigger point)之前或之后需要储存多少笔的跟踪记录。

注意: 当 HT-IDE3000 开始硬件仿真时（参考应用程序的硬件仿真章节），跟踪功能将会自动地开始记录被执行的指令与相关信息，反过来则不一定可以。

跟踪的初步设定

使用跟踪的基本要求，是在具有或是没有跟踪过滤条件下，设定跟踪模式(Trace Mode)。 跟踪模式定义程序的跟踪范围，而跟踪过滤条件(qualify)则是定义跟踪记录的条件。

可使用的跟踪模式有：

- Normal
  此为预设模式，设定跟踪的范围是全部的应用程序。
- Trace Main
  除了中断服务程序之外，其它所有的应用程序都属于跟踪的范围。
- Trace INT
  只有中断服务程序才属于跟踪的范围。

在硬件仿真过程中，跟踪会根据跟踪过滤条件决定哪条指令和何种相关的信息应该被记录在跟踪存储器内。规则是如果指令的相关信息与状态符合任何一个有效的跟踪过滤条件，则会记录这条指令。跟踪过滤条件的格式与断点的格式相同。如果需要将所有执行过的程序指令都记录起来，只要设定为 No Qualify 即可（也就是不要设定跟踪过滤条件），而这也是默认值。

相对于 Trace Mode 与 Qualify 为设定记录的条件，而 Trigger Mode 和 Forward Rate 则是指定停止跟踪记录的条件。

Trigger Mode 指定跟踪触发点的种类，它也可以用来决定停止跟踪点的位置。Forward Rate 则是设定跟踪范围的比率，此跟踪范围落在跟踪触发点与停止跟踪点之间。

可使用的 Trigger Mode 如下：

- No Trigger
  此为默认值，没有设定停止跟踪的条件。
- Trigger at Condition A
  触发点设在条件 A。
- Trigger at Condition B
  触发点设在条件 B。
- Trigger at Condition A or B
  触发点设在条件 A 或条件 B 两者之一。
- Trigger at Condition B after A
  触发点设在条件 B 但是要在条件 A 发生之后。
- Trigger when meeting condition A for k times
  触发点是当条件 A 符合 k 次时。
- Trigger at Condition B after meeting A for k times
  触发点是在条件 B 但是要在条件 A 符合 k 次后。

条件 A 和条件 B 设定触发条件，其格式和断点的格式相同。

Loop Count 用来设定条件 A 发生的次数，只有当 Trigger Mode 选用上述最后两个模式之一时才需要设定此项数字。

Forward Rate 用来设定跟踪记录段(trace record)占用跟踪存储器(trace buffer)的百分比，此段跟踪记录是介于跟踪触发点(trigger point)与停止跟踪点(stop trace point)之间的记录。从另外一个角度来看，可以先行设定跟踪触发点，再根据想要得到多少的跟踪数据，也就是跟踪触发点之后的数据，去设定 Forward Rate 百分比。跟踪触发点将跟踪存储器分为触发点之前与之后两部分，Forward Rate 设定触发点之后的百分比，以便限制跟踪记录的范围，百分比可设定在 0 至 100%之间。

注意: 跟踪记录的数据笔数不一定会刚好等于 Forward Rate。例如在跟踪记录尚未达到 Forward Rate 指定的百分比之前，碰到断点而停止仿真或是使用跟踪停止命令（参考停止跟踪的章节），这些都会停止跟踪记录，因此造成跟踪记录与 Forward Rate 的不一致。

Qualify 列表框记载及显示被 Trace Mode 使用到的跟踪过滤条件。列表框中可以加入高达 20 个过滤条件并且最多有 6 个过滤条件是同时有效的。可以将过滤条件设为无效或是把它从列表框中删除。在 Qualify 列表框中的跟踪过滤条件的格式与 Breakpoints 列表框中断点的格式相同（参考 Breakpoints 中的断点列表框格式部份）。

停止跟踪

有三种方法停止跟踪记录：

- 如上述，设定触发点（Trigger Mode）和 Forward Rate 百分比
- 设定断点去停止硬件仿真与跟踪记录
- 从 Debug 菜单（图 5-2）发出 Trace Stop 命令去停止跟踪记录

图 5-4 列出使用跟踪的必要数据，此对话框是执行 Debug 菜单中 Trace 命令的结果。

跟踪的启动与停止

→ **设定跟踪模式**

- 从 Debug 菜单选择 Trace 命令
  Trace 对话框显示如图 5-4。
- 从 Trace Mode 下拉式列表框选择跟踪模式
- 按下 OK 钮确认

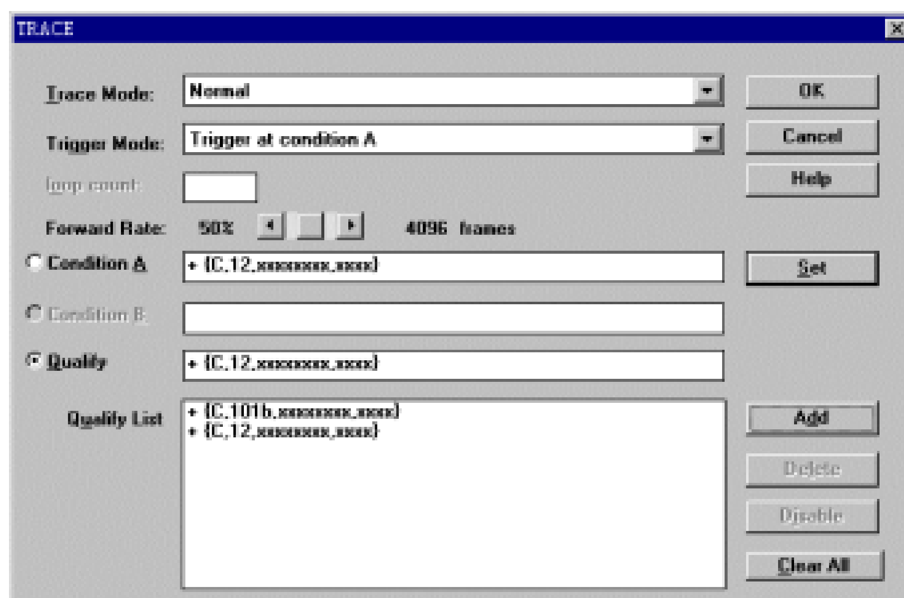


图 5-4

→ **设定停止跟踪的触发模式**

- 从 Debug 菜单选择 Trace 命令
  Trace 对话框会显示如图 5-4。
- 从 Trigger Mode 下拉式列表框选择触发模式
- 按下 OK 钮确认

→ **改变 Forward Rate**

- 从 Debug 菜单选择 Trace 命令
  Trace 对话框显示如图 5-4。
- 使用 Forward Rate 滚动条指定所需的百分比
- 按下 OK 钮确认

→ **设定条件 A 或条件 B**

- 从 Debug 菜单选择 Trace 命令
  Trace 对话框显示如图 5-4。
- 按下 Condition A 或 Condition B 钮
- 按下 Set Condition 钮
  Set Condition 对话框显示如图 5-5
- 键入条件 A 或条件 B 的数据

- 按下 OK 钮，关闭 Set Condition 对话框
- 按下 OK 钮，关闭 Trace 对话框

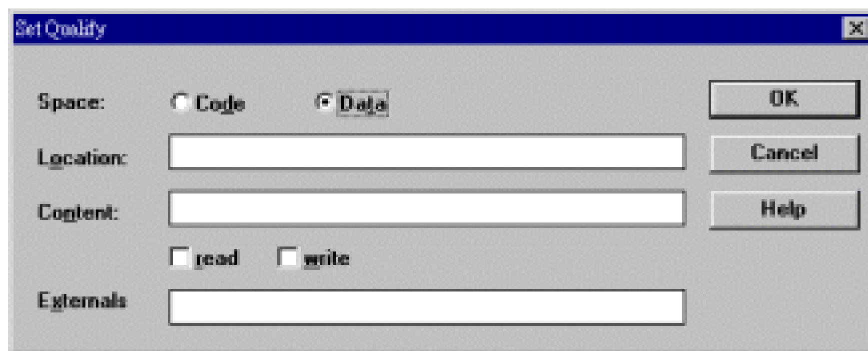


图 5-5

→ **加入跟踪的过滤条件**

- 从 Debug 菜单选择 Trace 命令
  Trace 对话框显示如图 5-4。
- 按下 Qualify 钮
- 按下 Set Qualify 钮
  Set Qualify 对话框显示如图 5-5
- 键入过滤条件的数据
- 按下 OK 钮，关闭 Set Qualify 对话框
- 按下 Add 钮，将过滤条件加入 Qualify List 框之内
- 按下 OK 钮，关闭 Trace 对话框

→ **删除跟踪的过滤条件**

- 从 Debug 菜单选择 Trace 命令
  Trace 对话框显示如图 5-4。
- 从 Qualify List 框中选择要被删除的过滤条件
- 按下 Delete 钮
- 按下 OK 钮确认

→ **删除所有的跟踪过滤条件**
- 从 Debug 菜单选择 Trace 命令
  Trace 对话框显示如图 5-4。
- 按下 Clear All 钮
- 按下 OK 钮确认

注意: 如果没有设定任何跟踪过滤条件，则所有指令将被默认为符合过滤条件。

→ **设定跟踪过滤条件为有效或无效**
- 从 Debug 菜单选择 Trace 命令
  Trace 对话框显示如图 5-4。
- 从 Qualify List 框中选择要被设定的过滤条件
- 按下 Enable（Disable）钮
- 按下 OK 钮确认

注意: 最多可同时设定 6 个跟踪过滤条件为有效。

跟踪记录的格式

在完成跟踪过滤条件和停止跟踪触发条件的设定之后，则所有符合过滤条件的指令将被记录在跟踪存储器内。使用 Window 菜单的 Trace List 命令可以查看跟踪记录数据，帮助程序的除错。当窗口中显示跟踪记录时，除了顺序编号一定会显示，跟踪记录内其它的数据并不一定都会显示出来，这些字段的显示与否取决于 Options 菜单中 Debug 命令的设定。在下面的说明标题文字中，用小括号括起来的文字会出现在 Window 菜单的 Trace List 命令标题上，代表各字段，例如顺序编号在 Trace 列表框的标题上是以 No. 来代表。图 5-6 和图 5-7 是在不同的 Debug 选项下所显示的跟踪列表的内容。

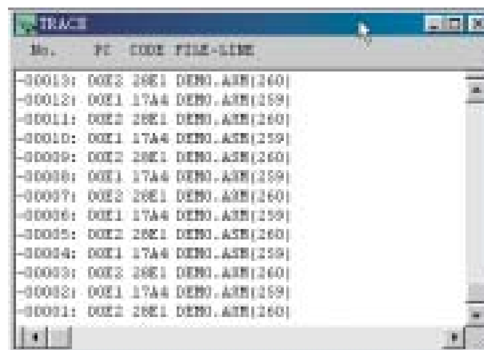


图 5-6

- 顺序编号（No.）
  不论何种停止跟踪触发模式，触发点的顺序编号均为+0，在触发点之前的跟踪记录使用负数的顺序编号，在触发点之后的跟踪记录则使用正数的顺序编号。如果跟踪记录中所有的字段（在 Options 菜单中的 Debug Option）都被圈选，则数据显示的结果如图 5-7。如果选择 No trigger mode 或是还未执行到触发点，则顺序编号是从-00001 开始往回以每笔跟踪记录减 1 的数字编列（图 5-6）。
- 程序计数器（PC）
  跟踪记录中指令的程序计数器值。
- 机器代码（CODE）
  指令的机器代码。
- 反编译指令（INSTRUCTION）
  反编译指令是执行 HT-IDE3000 的反编译程序所得的结果。
- 执行数据（DAT）
  被执行的数据内容（读或写）。
- 外部信号状态（0 1 2 3）
  外部信号 0~3 分别代表外部信号 ET0~ET3。
- 带有行号的源文件名称（FILE-LINE）
  源文件名称和此指令的行号。
- 源文件（SOURCE）
  源程序语句（包括符号）。

除了顺序编号一定会显示之外，上述中其它的字段均非必须的。

图 5-7

注意： 使用 Options 菜单中的 Debug 命令选择跟踪记录的字段。
使用 Window 菜单中的 Trace List 命令查看跟踪记录各字段的内容。

→ **清除跟踪存储器的内容**

使用 Debug 菜单的 Reset Trace 命令将跟踪存储器的内容清除，之后跟踪信息将由跟踪存储器的开始端储存起。另外，Reset 命令和 Power-On Reset 命令也会清除跟踪存储器的内容。

除错器的命令模式

除了窗口菜单的除错模式外，HT-IDE3000 另外提供一种称为命令式(Command Mode)的除错模式。在此模式下，除了具有窗口菜单除错模式的功能外，还提供额外的除错功能。这些增加的功能包括可以将除错的过程储存到记录文件，以后可以再次自动去执行这些除错命令，以及使用简易的命令语法，而不需重写整行命令即可执行先前的除错命令。

进入命令模式与离开命令模式

→ **进入命令模式**

从 HT-IDE3000 的 Debug 菜单中选择 “ Command Mode ” 命令，就会进入命令模式。此时会出现一个新的窗口画面，之后就可以在画面中第二行的提示字符 “ HT8> ” 之后，输入命令。(图 5-8)

→ **命令模式窗口**

- 命令模式的标题上会显示当前的项目名称。
- 在命令行提示符 “ HT8> ” 之后，可以输入任何命令。
- 当输入命令时，此命令的完整语法会显示在底下的状态栏中。
- 在提示字符串 “ HT8>xxxx ” 之后输入完整的命令后，下一行会显示命令执行的结果。(图 5-9)。
  再下一行则会显示提示字符串 “ HT8> ”，以等待另一个命令的输入。

→ **从命令模式离开**

离开命令模式的方式，可使用一般离开窗口的方法(按下窗口右上角的 x 框)，或是在提示字符串后输入 Q[uit]命令。

命令模式所支持的功能

下面的表格列出命令模式所支持的除错命令及其完整的语法。

命令	功能说明	命令语法
!	执行前一个命令	! dd
;	注释	;
BP	断点命令	BP{-C|-D|-E|-L}[list|*].→list=11 12…

命令	功能说明	命令语法
BP	断点设置	BP S[,RW],Location [,Data][,Ext Sig]
DB	显示程序存储器的内容	DB[bank.address[,range]]
DR	显示数据存储器的内容	DR[bank]address[,range]
FA	填入 ASCII 数据	FA {bank.address|symbol}list.→11 12…
FB	填入字节数据	FB {bank.address|symbol}list.→11 12…
GO	运行程序到指定的地址处	GO [address]
JP	直接跳跃到指定地址处	JP address
H	帮助	H
HIS	命令的历史记录	HIS
LF	载入并执行日志文件	LF [-V] [LogFileName]
LP	载入项目	LP ProjectName
Q	退出	Q
R	复位	R
POR	上电复位	POR
S	单步执行(Into/Over/Out)	S [-I|-V|-O] default option :“ -I ”
TR	跟踪列表	TR [-L][length]
W	打开/撰写/关闭日志文件	W {-S|-C}[LogFileName]

在除错命令语法中，若有大括号，则必须要在大括号中输入参数，否则会发生错误。其内的参数由 | 符号分隔。

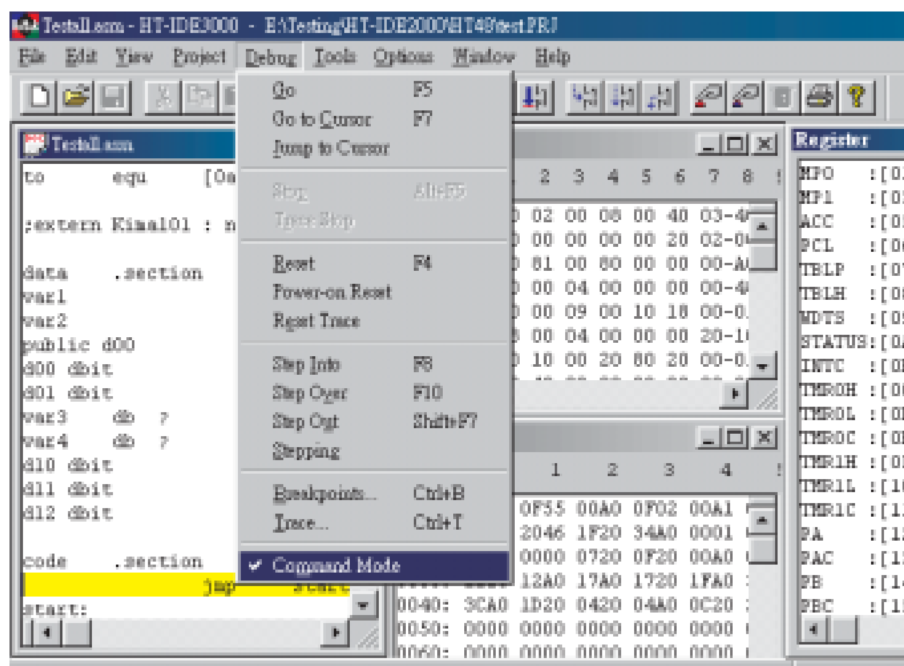


图 5-8

→ **断点命令**

有两种断点命令的格式，它们的命令语法和功能如下：

- BP - Breakpoint Clear/Enable/Disable/List

  语法：BP [-C | -D | -E | -L] [list |*]

  参数-C 是要清除断点。它会删除指定的断点或清除断点列表框内所有的断点。在列表框中的断点将从 1 到 20 编号，代表已经设置完毕的断点，可以一次选择多个断点。例如，三个由空格所分开的号码 1 3 8，是指第 1、3、8 个断点将被清除。它与 Debug/Breakpoint 窗口中的 Delete 钮有相同的功能。星号*表示要将所有已设好的断点清除掉，它与 Debug/Breakpoint 窗口中的 Clear All 钮有相同的功能。

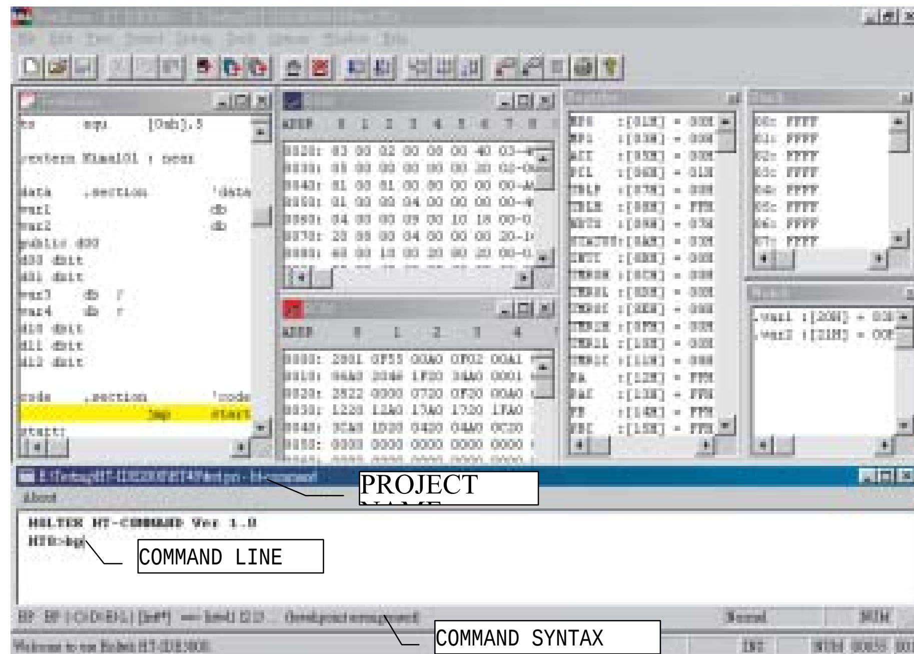


图 5-9

参数-D 会将所指定的断点改成无效状态，然而此断点仍然留在断点列表框内。此命令与 Debug/Breakpoint 窗口中的 Disable 钮有相同的功能。星号*的意义如上所述。

参数-E 会将所指定的断点改成有效状态，此命令与 Debug/Breakpoint 窗口中的 Enable 钮有相同的功能，星号*的意义如上所述。

参数-L 会将所有当前已设好的断点显示在窗口上，显示的格式与 Debug/Breakpoint 窗口的内容相同，而格式中第一列表示断点的编号。此编号可以应用在 BP -C，BP -D 及 BP -E 等命令中的 list 参数。

注意: 1. BP -L 不需要 list 参数。

2. 在任一时间内，HT-IDE3000 最多只能有 3 个断点是有效的。

如果没有给予 C、D、E 或 L 参数，则断点命令将会是以下的形式：

- BP – Breakpoint Set
  语法：BP S[,RW] ,Location [,Data][,Ext Sig]
  括号中的参数不一定都需要，然而在某些情况下必须要指定 。

  S 表示 Space，可在 C 或 D 之间选择一个。字母 C 表示断点设在程序存储器中，而 D 表示断点设定在数据存储器（RAM)。

  如果选择 D 取代 S，则必须再选定读取或写入的选项[,RW]，使用者可以选择 R 或 W 或 RW。这是因为如果断点设定在数据存储器中，则这个选择会让断点在读取数据，写入数据或读取写入数据时被激发。如果选择 C 取代 S，则表示程序代码，不需要设定 RW。

  " Location " 参数是设定断点的位置，其格式如下：
  [SourceFileName!].LineNumber 或[SourceFileName!].SymbolName
  如果没有指定源文件名称，则已经打开的源文件将被当成默认值。

  如果选择 D 取代 S，则必须设定 " Data " 参数。断点被设定在数据存储器中指定的位置，当发生读取或写入指定数据时触发中断。

  Ext Sig 是一个可以选择的参数，对于它的使用方法请查阅 HT-IDE3000 使用手册。

→ **注释命令**

- 语法：; comment string
  这个命令提供 Log 文件的说明格式。在; 之后的任何描述都不会被执行。

→ **显示命令**

- 语法：DB bank.address ,range

  DB range

  DB

  这个命令将所指定的程序存储器区域的内容显示在窗口中。这个区域是由地址，区域范围和存储器区块编号（bank number）所指定的，其格式为十六进制。如果指定了地址，但是没有指定存储器区块的编号，则会以当前的存储器区块的编号当做区块编号。如果地址及存储器区块的编号都没有指定，则会以当前的存储器区块的编号当做区块编号，以当前的程序计数器的内容当做地址。如果没有指定范围，则以 16 个字当做范围值，而范围不可以超过一个存储器区块的大小（2000h）。例如 1.0f00 表示存储器区块的编号为 1 而地址是 0f00h。

- 语法：DR bank.address ,range

  DR address

  这个命令会将所指定的数据存储器区域的内容显示在窗口中。这个区域是由地址，区域范围和存储器区块编号所指定，其格式为十六进制。如果没有指定范围，则被设为 16 个字节。范围大小不可以超过数据存储器区块的大小（100h）且区块编号要以十六进制表示。

→ **填充命令**

这个命令用来改变数据存储器的内容。

- 语法：FB {bank.address | symbol} ,list

  此命令会将 list 中的字节写到数据存储器中特定的地址，此地址是由存储器区块的编号及地址所指定或由符号来指定。可以使用 bank.address 或符号名称指定存储器的地址。 list 中的数据可以不只一个字节，但是在各字节之间必须至少有一个空格做为分隔。所有的数值都使用十六进制。list 的字节总个数不可超过存储器区块的大小。

- 语法：FA {bank.address | symbol} ,string

  FA 和 FB 有相同的功能，除了 string 是以 ASCII 标准码输入。

  使用者可以选用下列的任一种 symbol 格式：

  .var

  filename!.var

  path\ filename!.var

注意: 如果路径名称中包含有空格，则必须将该名称以引号括起来，否则会发生错误。

- 范例：FA " d:\tmp\test cmd\test1.asm!.count ", " test1 "

→ **执行命令/跳跃命令**

- 语法：GO [address]
  如果指定了地址，程序将会执行到这个指定的地址才停止。如果没有指定地址，程序将会执行到程序结束或碰到有效的断点才停止。
- 语法：JP address
  此命令会直接跳到所指定的地址，注意必须要指定地址。

→ **帮助命令**

- 语法：H
  这个命令会将所有的除错命令显示在窗口中，包括它们的语法和说明。

→ **历史记录命令**

- 语法：HIS
  此命令会在窗口中显示最近执行过的 20 条命令，但是不包括被执行的 HIS 命令。同时在每条命令的第一列会显示命令的编号。
- 语法：!dd
  dd 是上述 HIS 命令中被显示的命令编号，此命令将先前已执行过的命令再执行一次。以命令编号加上“！”的格式简化了需要重新输入完整的命令与参数的格式。如果没有指定命令的编号，则最后一个执行过的命令将被重新执行。

→ **载入命令**

- 语法：LF [-V] [LogFileName]
  此命令将 Log 文件的内容载入并且执行所载入的除错命令。LogFileName 是 Log 文件的文件名。
  如果没有指定 LogFileName，则会以当前的项目文件名称当做 LogFileName。

  参数 -V 是将命令行与命令执行的结果显示出来，
  如果 LF 没有设定 -V 选项，则执行的结果将会储存在记录文件中，而它的主文件名是取自 Log 文件的主文件名，后缀名为.res。

  Log 文件是写入命令(W)所生成的，可以使用 HT-IDE3000 的文件与编辑功能去修改其内容。这些内容必须是正确的除错命令，否则会发生错误并造成之后的除错命令不能被执行并返回到提示符。

注意: 1. 如果 LogFileName 中包含空格，则必须使用引号将名称括起来，否则会发生错误。
2. Log 文件不能含有 LF、W 或 Q 命令。

→ **退出命令**

- 语法：Q

  此命令将结束命令模式并且回到先前的窗口。

注意: 1. 此命令在 Log 文件中无效。

2. 从命令模式退出之后，所有通过“ LF ”和“ W -S ”命令所打开的文件都将被关闭，并且停止执行命令。

→ **复位命令**

- 语法：R

  此命令的功能与 Debug 菜单的 Reset 命令相同。

- 语法：POR

  此命令的功能与 Debug 菜单的 Power-On Reset 命令相同。

→ **单步执行命令**

有三种单步执行的命令，每当执行完毕均会显示 PC、STATUS 或 ACC 的内容。

- 语法：S {-I | -O | -V}

  单步执行命令。

  -I 是 Step Into，与 Debug 菜单的 Step Into 命令有相同的功能。

  -O 是 Step Over，与 Debug 菜单的 Step Over 命令有相同的功能。

  -V 是 Step Out，与 Debug 菜单的 Step Out 命令有相同的功能。

  如果没有设定选项，则系统默认值为“ S -V ”。

→ **跟踪命令**

- 语法：TR [-L] [length]

  跟踪命令会将跟踪存储器的内容显示在窗口中，参数-L 是要显示跟踪记录内所有的字段，包括顺序编号、程序计数器、机器代码、反编译指令、执行数据、外部信号、源文件名称、行号和源程序。

  如果没有指定参数 -L，则只会显示顺序编号、程序计数器、机器代码、反编译指令和源文件名称及行号。参数“ length ”指定要显示多少笔跟踪记录。从编号为 0 的跟踪记录开始往回到所指定的长度，也可以向前跟踪所指定的长度。如果要向前跟踪，必须要预先设定 Forward Rate。系统预设长度值为 5。

命令模式下不能设定跟踪模式、跟踪过滤条件和停止跟踪百分比等参数，必须要在 HT-IDE3000 的窗口中设定。

→ **写入命令**

- 语法：W [-S｜-C] [LogFileName]

  此指令会将除错命令与其相对应的执行结果写入 Log 文件。当执行 W -C 命令或 Q 命令或是终止命令模式后，便会停止写入 Log 檔。

  参数 -S 生成一个 Log 文件，而所有接下来的命令与结果会被写入此文件。

  参数 -C 会关闭先前所生成的 Log 文件，而且不会再有命令被写入 Log 文件。

  如果 Log 文件已经存在，则系统会要求确认是否同意要覆盖原先的数据并继续下一步。指定 LogFileName 文件名时，不需要加上后缀名。

  如果 LogFileName 指定的 Log 文件不存在，则会使用项目名称当做 Log 文件的主文件名，而以.CMD 当做后缀名。

注意:
1. 如果 LogFileName 文件名中含有空格，则必须以引号将文件名括起来，否则会发生错误。
2. 在 W -S 命令执行之后，不可以使用 LF 命令或 W -S 命令。

Log 文件格式

Log 文件是一个可以被任何文字编辑器，包括 HT-IDE3000 在内的编辑器所撰写及修改的文本文件。可以使用 Edit 菜单去撰写，其格式为每一条除错命令占用一行。

命令 W -S LogFileName 将会清除 Log 文件的内容，之后写入新的命令及执行结果。

如果命令字符串是由 W -S 命令所生成的，则提示符也会一并写入 Log 文件中，但是下一次从 Log 文件中读取除错命令时，先前被写入的提示符将自动地被忽略。如果是使用编辑器生成命令字符串，则不需要将提示符写到 Log 文件。

如果 Log 文件是由命令“ W -S ”所生成的，则会在每一个命令执行结果的文字行最前面，自动插入一个“ ;”代表注释。
当下一次载入 Log 文件并执行其中的除错命令时，只有命令字符串行会被执行，而执行结果字符串行将被忽略。

HT-COMMAND 错误信息

错误信息	说明
Invalid Command	键入的命令不正确
Can not find HT-IDE	现在的环境不是 HT-IDE3000
Syntax error	输入的语法不正确
No project for debug	在 HT-IDE3000 中没有打开的项目文件
ROM bank Out of range	指定显示的 Program Memory 超出范围
RAM bank Out of range	指定显示的 Data Memory 超出范围
Can not run xxx command in emulation mode	在仿真模式中，执行命令 xxx 失败
Can not run xxx command in load file mode	在下载文件模式中，执行命令 xxx 失败
Can not run xxx command in write file mode	在 write file 模式中，执行命令 xxx 失败
Unterminated string	字符串需要对称的括号
No Command in history buffer	History buffer 中没有数据
Open xxx log file error	无法打开 Log 文件
Close xxx log file error	无法关闭 Log 文件
Read xxx log file error	无法读取 Log 文件
Write xxx log file error	无法写入 Log 文件
Not in emulation status	执行本命令前需先进入 emulation mode
Sources have been modified，please rebuild	原先的源文件已被修改过，需重新编译项目
Stop by user	使用者强迫停止执行
Get PC failed	无法读取 Program Counter 的值
Stack overflow	堆栈超出其容量
No debug info	设定的断点没有除错信息
Cannot find the symbol	找不到指定的符号
Cannot find the register	找不到指定的寄存器

第六章

菜单 – 窗口

6

HT-IDE3000 提供许多种类的窗口，以便帮助使用者对应用程序进行硬件或软件仿真，这些窗口(如图 6-1 所示)包括数据存储器(RAM)、程序存储器(ROM)、跟踪列表、寄存器内容、变量检测、堆栈、程序、命令执行等数据窗口。

图 6-1

窗口菜单命令

- RAM
  RAM 窗口显示数据存储器空间的内容，如图 6-2 所示。专用寄存器的地址空间并不包含于 RAM 窗口中，因为它会显示在寄存器窗口中。RAM 窗口的内容可以因除错的目的而直接被修改，垂直方向显示的地址是基本地址而水平方向各数字则是地址偏移量，所有的数字均以十六进制格式显示。

图 6-2

- ROM
  ROM 窗口显示程序存储器空间的内容，如图 6-3 所示。ROM 地址范围是从 0 到最大的程序地址，此最大地址取决于在项目中所选取的单片机。水平与垂直滚动条可以用来查看在 ROM 窗口中任意的地址，ROM 窗口的内容以十六进制格式显示并且不可以被修改。

图 6-3

- Trace List

  Trace List 窗口显示跟踪记录的信息，如图 6-4 所示。跟踪记录的内容可在 Options 菜单中 Debug 命令定义。使用鼠标双击 Trace List 窗口中的跟踪记录会将此行所在的源程序文件打开并且放置于工作的源文件窗口中，光标停在对应的行上。

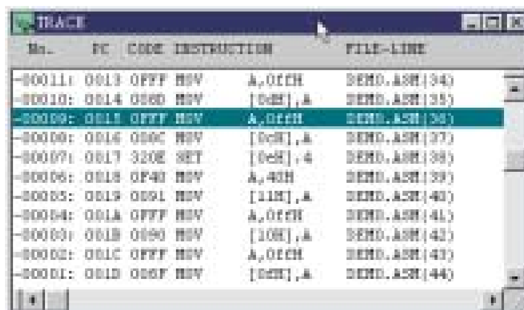


图 6-4

- Register

  Register 窗口显示项目中所选取的单片机内所有被定义的寄存器内容，图 6-5 所示为 HT48C70-1 的寄存器窗口的范例，Register 窗口的内容可被修改，此窗口是可随处停留的，可以在主窗口内把它到处移动。

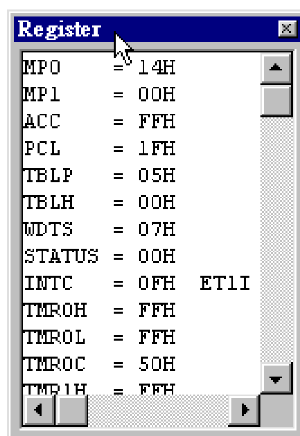


图 6-5

- Watch

  Watch 窗口显示所指定符号的存储器地址和内容，此符号是定义在数据存储器，即 RAM 空间中的。符号的格式如下：

  ```
  [source_file_name!].symbol_name
  ```

  可以显示寄存器的内容，只要先键入句点(.)然后键入符号名称或寄存器名称再按下 Enter 键。指定的符号或寄存器的存储器地址及内容将被显示在符号的右边，如下列格式：

  ```
  :[address]=data contents
  ```

  地址与数据两者都以十六进制格式显示，如图 6-6 所示。符号与相对应的数据会被 HT-IDE3000 储存并且在下一次打开 Watch 窗口时显示出来。按下 Delete 键可将符号从 Watch 窗口中删除，Watch 窗口也是可以随处移动的。

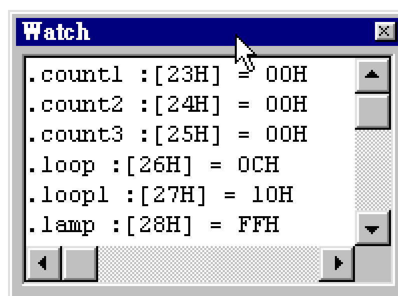


图 6-6

- Stack

  Stack 窗口显示当前项目中所选择单片机堆栈寄存器的内容，最大的堆栈层数是根据所选的单片机规格而定。图 6-7 显示一个 Stack 窗口的例子。 堆栈是从 0 开始往上增加层次，每次入栈操作会使数字加 1（例如 CALL 指令或中断响应），而每次出栈操作则使它减 1（RET 或 RETI 指令）。最上层的堆栈会以高亮标示，例如图 6-7 中 01 表示最上层的堆栈行。当执行一个 RET 或 RETI 指令时，标示在最上层的程序行号（此例中为 134）被视为下一个要执行的指令，同样在堆栈行的最上层的再上一行（此例中为 00）将被当成新的最上层堆栈行。如果堆栈中没有任何堆栈行，则 Stack 窗口中将没有任何行会被高亮标示。堆栈行的格式如下：

  ```
  Stack_level: program_counter source_file_name(line_number)
  ```

此处 stack_level 是堆栈的层数，program_counter 为调用程序的十六进制返回地址或中断时的程序地址，source_file_name 是包含此调用程序或 0 中断指令的源文件名称，而 line_number 则是在源程序文件中调用指令或中断响应之后一条指令的行数，line_number 是十进制数。

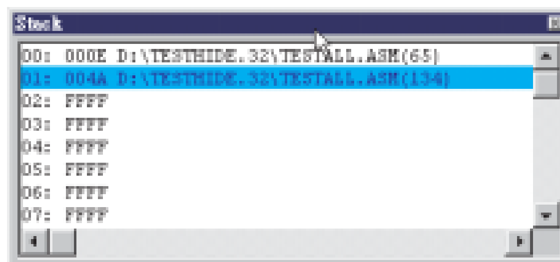


图 6-7

- Program

  Program 窗口以反编译的格式显示程序存储器即 ROM 的内容。地址范围是从 0 到最大的地址为止，此最大的地址由项目中所选取的单片机决定。
- Output

  Output 窗口显示执行 Build 或 Rebuild All 命令后的结果信息。在错误信息行双击鼠标左键，会出现包含源程序文件的窗口，并且错误所在程序行会被高亮标示。

第七章

软件仿真

7

HT-IDE3000 除了硬件仿真之外，同时提供软件仿真的方法以支持程序的除错。软件仿真器所支持的功能与 HT-ICE 提供的相同，但不需要使用 HT-ICE。在 HT-IDE3000 中 HT-ICE 的除错功能和窗口显示功能对软件仿真器同样有效，此外软件仿真器还提供一个输入输出使用接口。虽然软件仿真器提供了许多除错功能，但单片机的某些硬件特性仍然无法仿真，因此在生产制造掩膜型 IC 之前，建议还是先要使用 HT-ICE 执行应用程序的硬件仿真。

某些单片机系列只支持硬件仿真模式，而有些则同时支持硬件与软件仿真模式。

开始仿真

进入 HT-IDE3000 之后，有两种情况会发生，一是项目已经被打开，另一种是没有项目被打开。在第一种的情况下，HT-IDE3000 的工作模式取决于此项目的工作模式，而在第二种情况下，则是软件仿真模式。即使项目的工作模式为硬件仿真，使用者仍可自行更改为软件仿真，另外当下列的情况发生时，HT-IDE3000 的工作模式将是软件仿真。

- HT-ICE 和主机没有连接或连接失败时
- HT-ICE 电源关闭时

在 Options 菜单的 Debug 命令中可以设定 HT-IDE3000 的工作模式，图 7-1 显示 Debug 命令的内容。

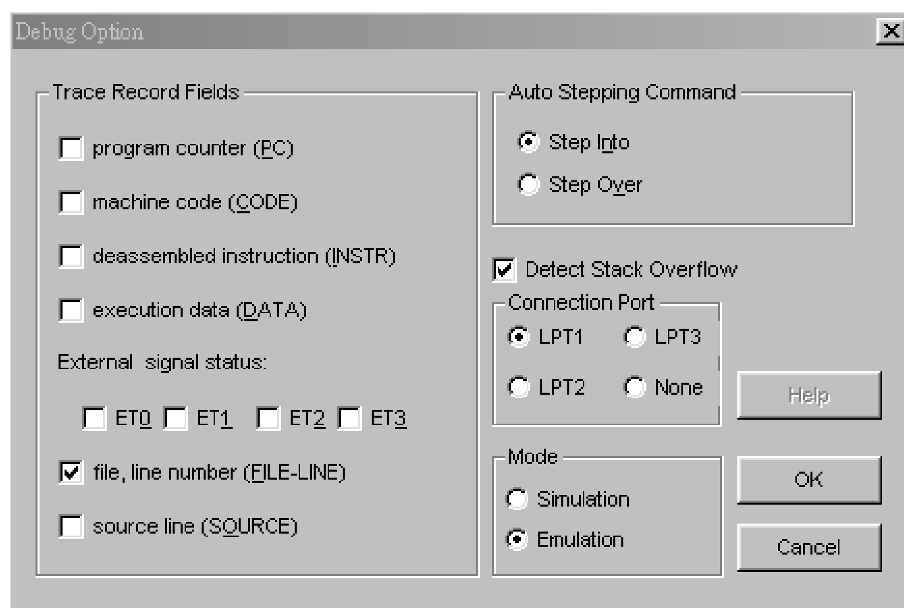


图 7-1

除了软件仿真器之外，盛群还提供虚拟外围器件管理（VPM），能够让使用者直接在 PC 上驱动与监控输入和输出的仿真。

在第三部份中将针对 VPM 做更详细的介绍。

第八章

OTP 烧录

8

简介

OTP writer 是一种专为烧录 OTP（One-Time Programmable 一次烧录）单片机的简易烧录器。凡是盛群半导体公司开发完成的这类 OTP 单片机，都可使用简易烧录器将程序数据烧录到芯片内。此烧录器的特点为轻巧短小，同时安装及使用都很容易，功能简单明了。最新版本的 HT-ICE 仿真器则更进一步将此烧录器整合到 HT-ICE 仿真器上，使用者在产品开发上将会更得心应手。如图 8-1 所示：

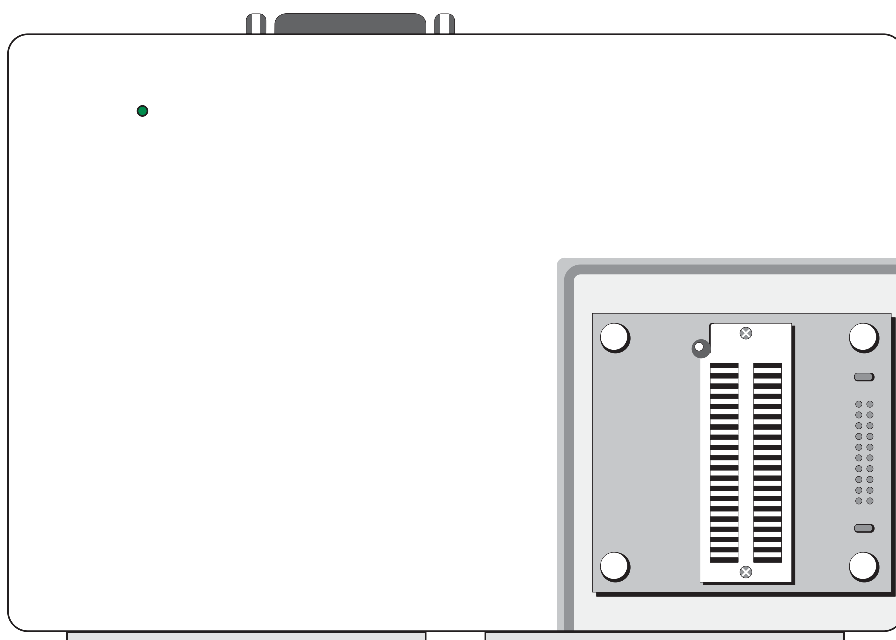


图 8-1

安装

由于此烧录器已整合到 HT-ICE 仿真器上，如果使用者已将 HT-ICE 安装完毕，则在 HT-IDE3000 软件中直接启动烧录功能即可,不再需要执行其它安装程序。HT-IDE3000 软件系统与 HT-ICE 仿真器的安装步骤，请参考第一章概要与安装。

烧录器适配卡

HT-ICE 仿真器出厂时已配置 40-pin TEXTOOL 烧录器适配卡（图 8-2）。如果使用者所需烧录的单片机封装无法适用时，须使用者自行更换烧录器的适配卡。HT-ICE 烧录器适配卡进一步的信息，请参考其他相关技术文件或至本公司网站查询。

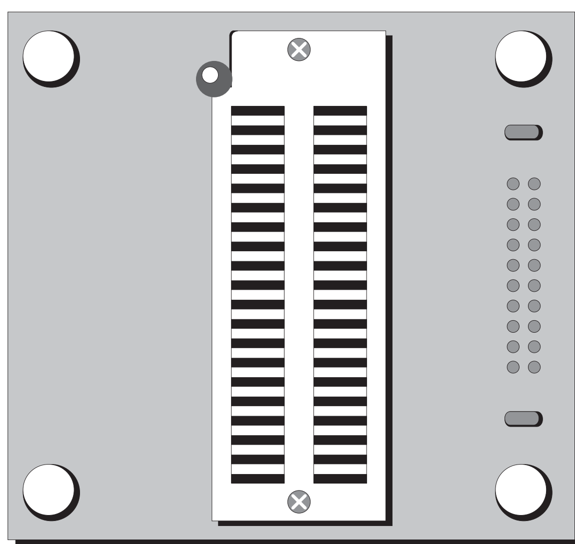


图 8-2

使用 HT-HandyWriter 烧写 OTP 单片机

→ **启动 HT-HandyWriter 烧录程序**

选用主窗口中程序菜单的 Holtek Development System，然后执行小图标 HT-HandyWriter，如下图 8-3 所示：

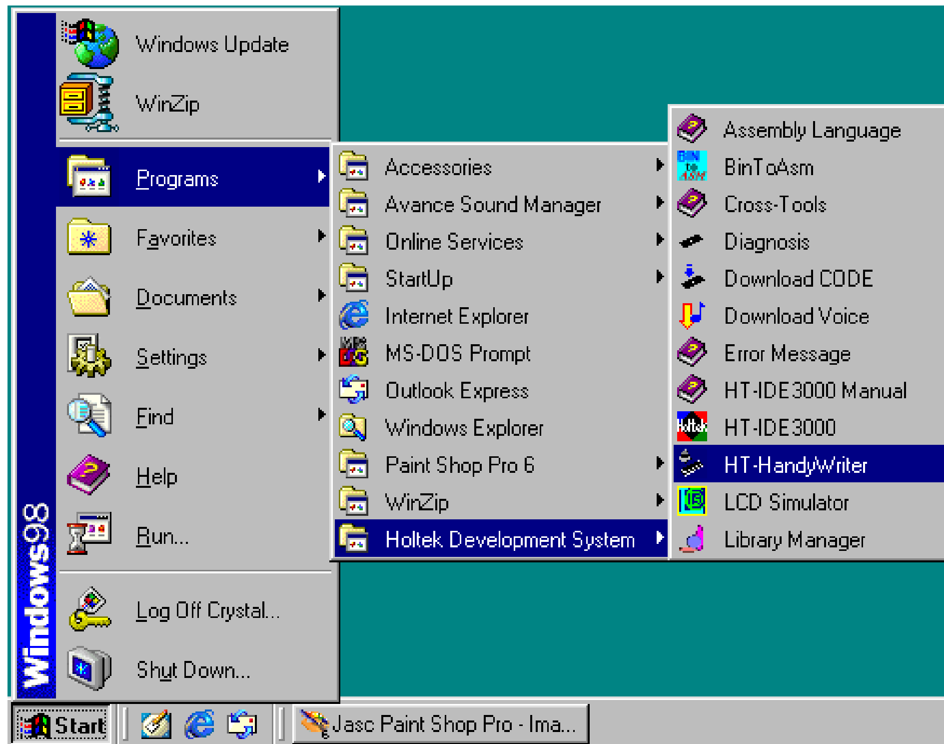


图 8-3

→ **LPT－设定打印机口**

启动 HT-HandyWriter 进入图 8-4 的窗口后，需要事先设定打印机口。使用鼠标点击 " LPT " 菜单选项，出现图 8-5 的打印机口菜单。可以从 LPT1、LPT2、LPT3 中选用一个。如果 OTP writer 是连接到 HT-ICE，则视 HT-ICE 与个人计算机的第几个打印机口相连而定，例如 HT-ICE 与个人计算机的 LPT1 相接，则在图 8-5 中选择 LPT1。如果 OTP writer 是直接与个人计算机的打印机口相连，则设定此相连的打印机口即可。

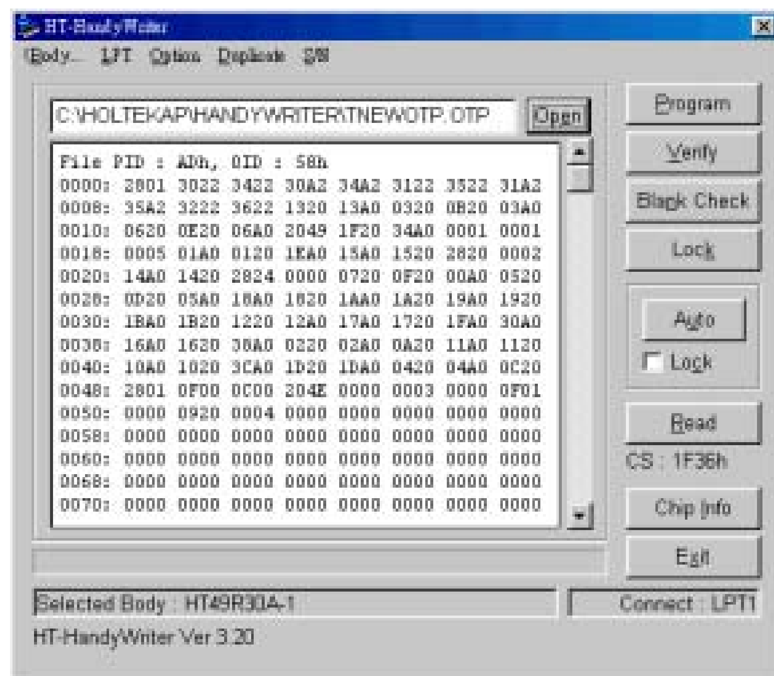

HT-HandyWriter
Body... LPT Option Duplicate S/N
C:\HOLTEKAP\HANDYWRITER\TNEWOTP.OTP
Open
File PID : ADh, OID : 58h
0000: 2801 3022 3422 30A2 34A2 3122 3522 31A2
0008: 35A2 3222 3622 1320 13A0 0320 0B20 03A0
0010: 0620 0E20 06A0 2049 1F20 34A0 0001 0001
0018: 0005 01A0 0120 1EA0 15A0 1520 2820 0002
0020: 14A0 1420 2824 0000 0720 0F20 00A0 0520
0028: 0D20 05A0 18A0 1820 1AA0 1A20 19A0 1920
0030: 1BA0 1B20 1220 12A0 17A0 1720 1FA0 30A0
0038: 16A0 1620 38A0 0220 02A0 0A20 11A0 1120
0040: 10A0 1020 3CA0 1D20 1DA0 0420 04A0 0C20
0048: 2801 0F00 0C00 204E 0000 0003 0000 0F01
0050: 0000 0920 0004 0000 0000 0000 0000 0000
0058: 0000 0000 0000 0000 0000 0000 0000 0000
0060: 0000 0000 0000 0000 0000 0000 0000 0000
0068: 0000 0000 0000 0000 0000 0000 0000 0000
0070: 0000 0000 0000 0000 0000 0000 0000 0000
Program
Verify
Blank Check
Lock
Auto
Lock
Read
CS : 1F36h
Chip Info
Exit
Selected Body : HT49R30A-1
Connect : LPT1
HT-HandyWriter Ver 3.20

图 8-4

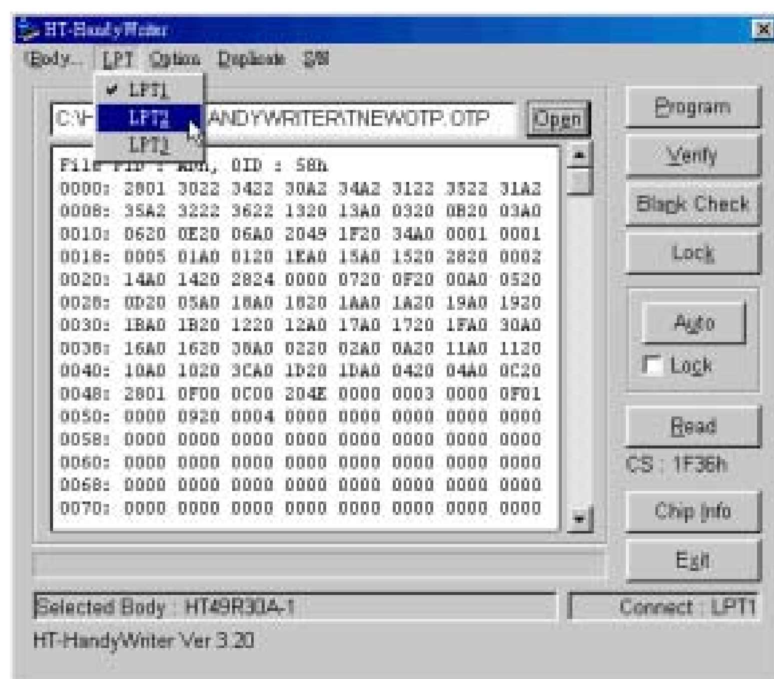

HT-HandyWriter
Body... LPT Option Duplicate S/N
LPT1
LPT2
LPT3
ANDYWRITER\TNEWOTP.OTP
Open
0000: 2801 3022 3422 30A2 34A2 3122 3522 31A2
0008: 35A2 3222 3622 1320 13A0 0320 0B20 03A0
0010: 0620 0E20 06A0 2049 1F20 34A0 0001 0001
0018: 0005 01A0 0120 1EA0 15A0 1520 2820 0002
0020: 14A0 1420 2824 0000 0720 0F20 00A0 0520
0028: 0D20 05A0 18A0 1820 1AA0 1A20 19A0 1920
0030: 1BA0 1B20 1220 12A0 17A0 1720 1FA0 30A0
0038: 16A0 1620 38A0 0220 02A0 0A20 11A0 1120
0040: 10A0 1020 3CA0 1D20 1DA0 0420 04A0 0C20
0048: 2801 0F00 0C00 204E 0000 0003 0000 0F01
0050: 0000 0920 0004 0000 0000 0000 0000 0000
0058: 0000 0000 0000 0000 0000 0000 0000 0000
0060: 0000 0000 0000 0000 0000 0000 0000 0000
0068: 0000 0000 0000 0000 0000 0000 0000 0000
0070: 0000 0000 0000 0000 0000 0000 0000 0000
Program
Verify
Blank Check
Lock
Auto
Lock
Read
CS : 1F36h
Chip Info
Exit
Selected Body : HT49R30A-1
Connect : LPT1
HT-HandyWriter Ver 3.20

图 8-5

→ **！Body－选取 OTP MCU 类型**

使用鼠标点击“ !Body ”命令，出现图 8-6 所示的[Set Body]对话框。若 OTP 内部未记录单片机类型识别码的话，则以使用者所选的单片机类型为准，完成所有写入与读取动作。

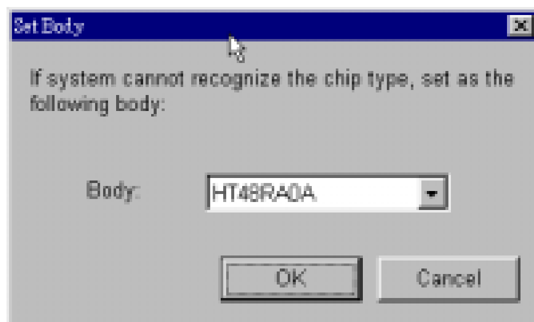


图 8-6

→ **选项－检查 MCU 选项**

- Option

  当选择[Option]/Option 命令时，将会出现如图 8-7 的对话框，它会将已打开的文件或 OTP writer 上的 OTP 单片机的掩膜选项显示出来。

- Print

  此命令将会将已打开文件或 OTP 单片机的掩膜选项打印出来。

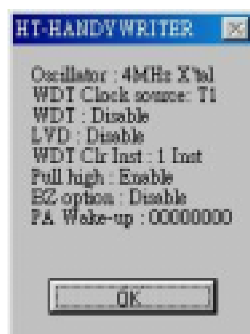


图 8-7

→ **HT-HandyWriter 烧录功能（Programming Functions）**

图 8-4 显示 HT-HandyWriter 提供的内部功能，在 HT-HandyWriter 窗口内的右边有 9 个按钮（buttons），每个按钮代表一个命令，各命令功能说明如下：

- Open
  打开一个后缀名为.OTP 的文件，并将其内容载入到计算机的内存，当执行烧录时会从此处取得数据。按下 Open 钮之后，会出现目录与文件名可供选定。文件打开后，文件内容会显示在信息窗口内，并在 Read 按钮下方显示所打开文件的校验码（Checksum）。
- Program
  此命令会执行两项功能，首先会将计算机内存中的数据烧录到 OTP writer 上的 OTP 单片机内，之后再比较 OTP 单片机的内容是否与计算机内存的数据一致，并将结果显示在 HT-HandyWriter 窗口内。
- Verify
  验证 OTP 单片机的内容是否与当前计算机内存的数据一致。先从 OTP 单片机读取内容再做比较，验证结果会显示在 HT-HandyWriter 窗口内。
- Blank Check
  检查 OTP writer 上的 OTP 单片机是否已经被烧写过。检查的结果显示在窗口内，如果不是空的 IC，已被烧写过的位置会显示出来。
- Lock
  此命令会使 OTP writer 设置 OTP 单片机的保护功能，禁止将此颗 OTP 单片机的内容读出。一般是在执行 Program 功能之后，需要将 OTP 单片机的内容做保护时，使用此命令。
- Auto
  此命令会以 Blank Check、Program 与 Verify 的顺序将此三个命令执行完毕，若任何命令的执行有错，将会停止并且不再继续执行下一个命令。底下有一个 Lock 的选项，可以防止数据在烧录后被读取。如果需要做防止读取，则必须先选取此项，然后再按下 Auto 钮。
- Read
  此命令会将 OTP writer 上 OTP 单片机的内容读出，再将之存入计算机的内存内，并在 Read 按钮下方显示文件内容的校验码（Checksum）。也可以把它存入后缀名为.OTP 的文件内。

- Chip Info

  此命令会从单片机内读出 Power-On ID，Software ID，ROM size，Option size 等信息，并显示 " Get Info from chip "，告知使用者以上信息是从单片机中取得。若单片机内部没有记录这些信息的话，则会显示由!Body 命令所设定的单片机规格，并显示 " Get Info from ini "，告知使用者以上信息由系统的设定取得。

→ **HT-HandyWriter 其它功能**

- Duplicate－自动侦测 OTP 单片机存在与否及连续烧录

  Duplicate 提供可以连续烧录多颗同一型号 OTP 的功能。当使用上述 HT-HandyWriter 烧录功能的 Open 命令打开.OTP 文件后，再将尚未烧录的 OTP 单片机放入 TEXTOOL 并夹住，HT-HandyWriter 便会自动侦测，同时自动执行所设定的功能。以这种方式，只要执行一次打开.OTP 文件的动作，之后只需要重复更换 OTP 单片机，即可连续执行所设定的功能。

  在使用此功能之前，首先设定需要连续自动执行的功能，其方式是点击如图 8-8 所示的[Duplicate]/Setup 命令，即可以设定连续自动执行的功能。如图 8-9 所示的 Duplicate Setup 画面提供使用者选取包括 Blank Check，Program，Verify，Lock 等功能。

  接着需要点击如图 8-10 所示的[Duplicate]/Enable 指令，以启动连续自动执行的功能。当完成启动之后，就可以连续自动地大量烧录 OTP 单片机。当所有 OTP 单片机烧录完成，不再需要连续自动地烧录功能时，只需要使用鼠标点击如图 8-10 所示的[Duplicate]/Enable 指令，则会中止连续自动执行的功能。

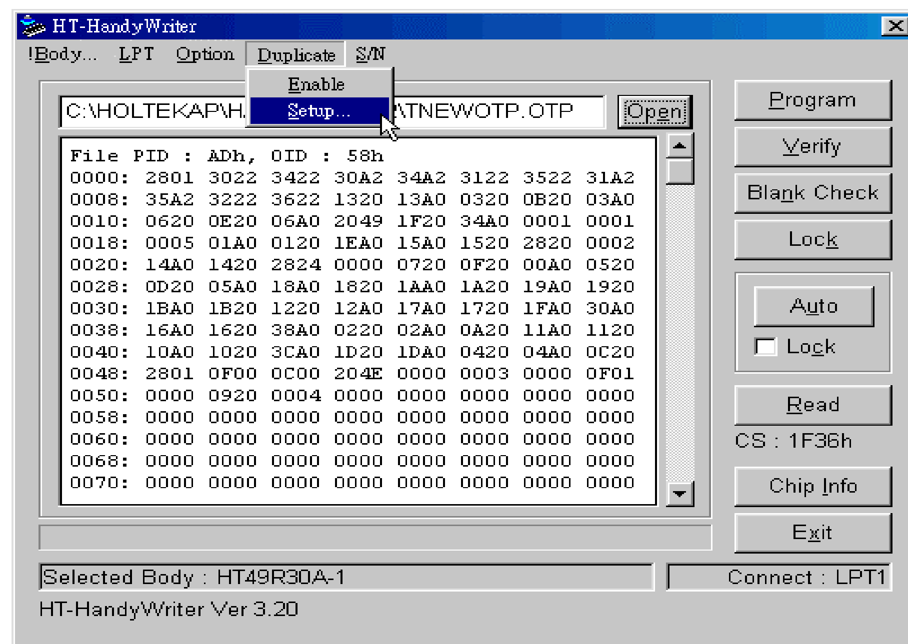

HT-HandyWriter
!Body... LPT Option Duplicate S/N
Enable
Setup...
C:\HOLTEKAP\H.
\TNEWOTP.OTP
Open
File PID : ADh, OID : 58h
0000: 2801 3022 3422 30A2 34A2 3122 3522 31A2
0008: 35A2 3222 3622 1320 13A0 0320 0B20 03A0
0010: 0620 0E20 06A0 2049 1F20 34A0 0001 0001
0018: 0005 01A0 0120 1EA0 15A0 1520 2820 0002
0020: 14A0 1420 2824 0000 0720 0F20 00A0 0520
0028: 0D20 05A0 18A0 1820 1AA0 1A20 19A0 1920
0030: 1BA0 1B20 1220 12A0 17A0 1720 1FA0 30A0
0038: 16A0 1620 38A0 0220 02A0 0A20 11A0 1120
0040: 10A0 1020 3CA0 1D20 1DA0 0420 04A0 0C20
0048: 2801 0F00 0C00 204E 0000 0003 0000 0F01
0050: 0000 0920 0004 0000 0000 0000 0000 0000
0058: 0000 0000 0000 0000 0000 0000 0000 0000
0060: 0000 0000 0000 0000 0000 0000 0000 0000
0068: 0000 0000 0000 0000 0000 0000 0000 0000
0070: 0000 0000 0000 0000 0000 0000 0000 0000
Program
Verify
Blank Check
Lock
Auto
Lock
Read
CS : 1F36h
Chip Info
Exit
Selected Body : HT49R30A-1
Connect : LPT1
HT-HandyWriter Ver 3.20

图 8-8

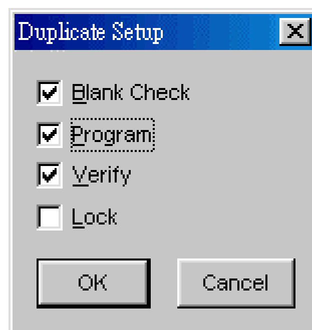

Duplicate Setup
Blank Check
Program
Verify
Lock
OK
Cancel

图 8-9

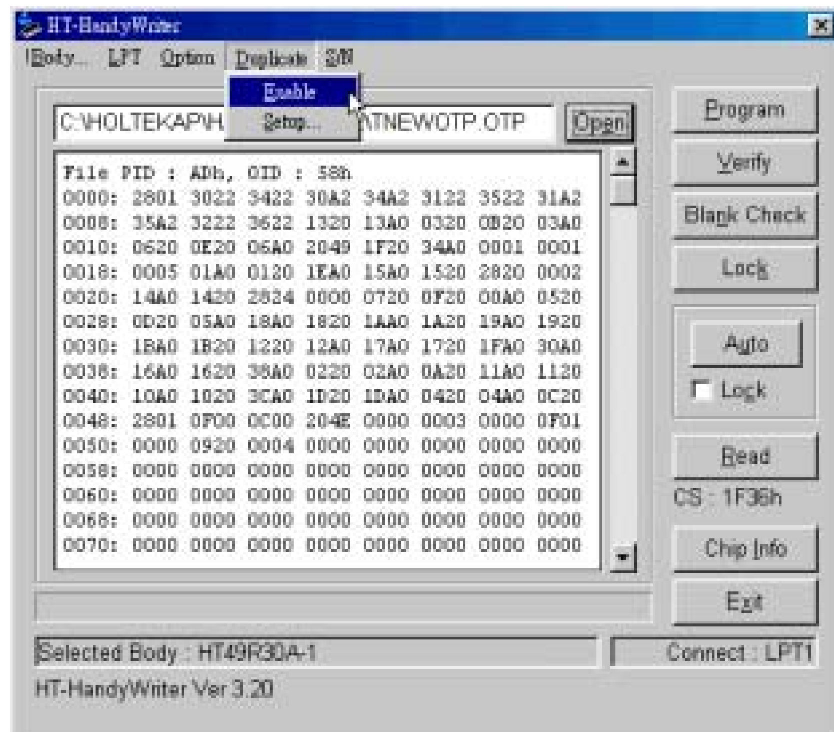


图 8-10

- S/N一序列号（Serial Number）的写入

  序列号的功能是可以将指定的序列号烧写到单片机内。此序列号和它的地址是由使用者指定然后烧录到每个单片机的程序存储器的低字节，而且在烧录完成之后，序列号会自动加一，以便供下一个要烧录的单片机使用。

  首先需要设定开始的序列号和烧录的存储器地址，使用鼠标点击[S/N]菜单的 Setup 命令即出现如图 8-11 所示的 Serial Number 窗口，即可输入开始的序列号和地址。

  完成初始值的设定后，接着使用鼠标点击[S/N]/Enable 命令以便启动此功能，而如图 8-12 所示，在主窗口的右下角会显示出当前序列号的相对应地址和数据。在烧录第一颗单片机的过程中，便会将此序列号写进所指定的程序存储器地址内，接下来烧录的单片机的序列号则会自动加一。若要重新设定序列号，则再点击如图 8-12 所示的[S/N]菜单的 Setup 命令。

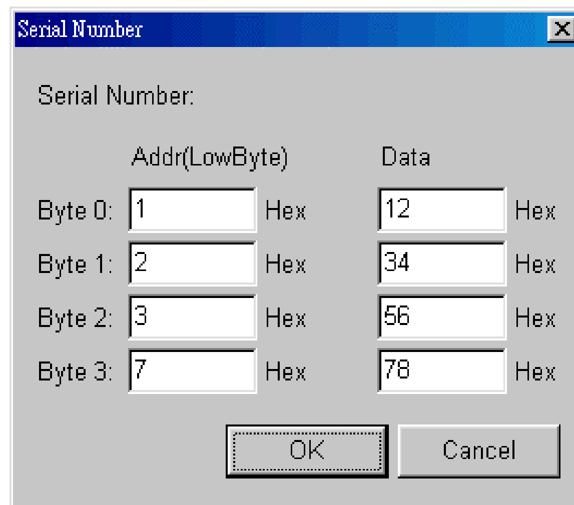

Serial Number
Serial Number:
Addr(LowByte)
Data
Byte 0: 1 Hex 12 Hex
Byte 1: 2 Hex 34 Hex
Byte 2: 3 Hex 56 Hex
Byte 3: 7 Hex 78 Hex
OK
Cancel

图 8-11

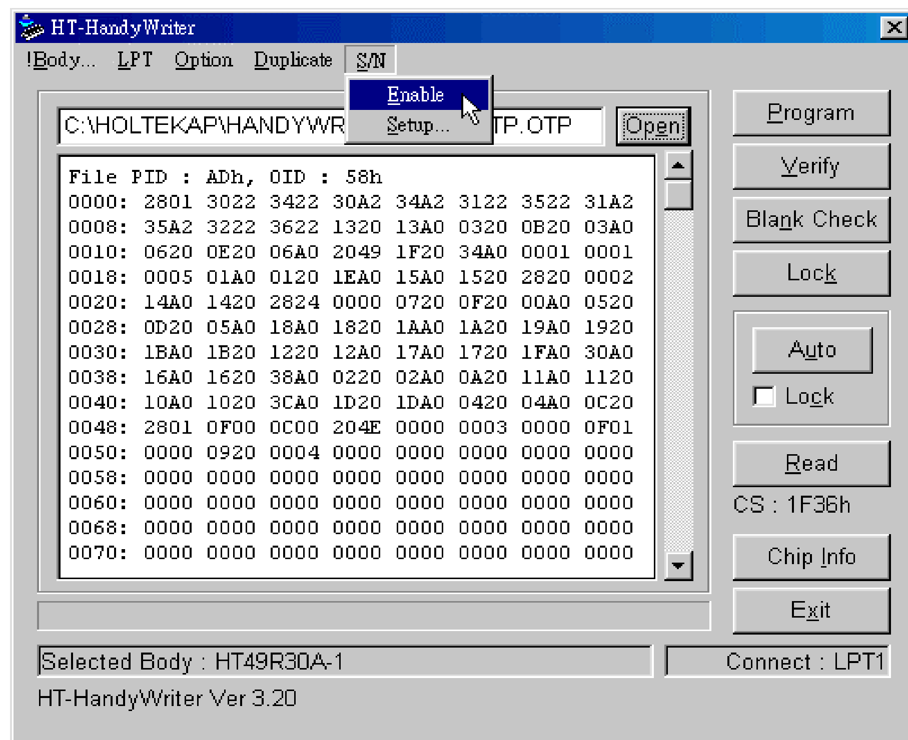

HT-HandyWriter
!Body... LPT Option Duplicate S/N
Enable
Setup...
C:\HOLTEKAP\HANDYWR
TP.OTP
Open
File PID : ADh, OID : 58h
0000: 2801 3022 3422 30A2 34A2 3122 3522 31A2
0008: 35A2 3222 3622 1320 13A0 0320 0B20 03A0
0010: 0620 0E20 06A0 2049 1F20 34A0 0001 0001
0018: 0005 01A0 0120 1EA0 15A0 1520 2820 0002
0020: 14A0 1420 2824 0000 0720 0F20 00A0 0520
0028: 0D20 05A0 18A0 1820 1AA0 1A20 19A0 1920
0030: 1BA0 1B20 1220 12A0 17A0 1720 1FA0 30A0
0038: 16A0 1620 38A0 0220 02A0 0A20 11A0 1120
0040: 10A0 1020 3CA0 1D20 1DA0 0420 04A0 0C20
0048: 2801 0F00 0C00 204E 0000 0003 0000 0F01
0050: 0000 0920 0004 0000 0000 0000 0000 0000
0058: 0000 0000 0000 0000 0000 0000 0000 0000
0060: 0000 0000 0000 0000 0000 0000 0000 0000
0068: 0000 0000 0000 0000 0000 0000 0000 0000
0070: 0000 0000 0000 0000 0000 0000 0000 0000
Program
Verify
Blank Check
Lock
Auto
Lock
Read
CS : 1F36h
Chip Info
Exit
Selected Body : HT49R30A-1
Connect : LPT1
HT-HandyWriter Ver 3.20

图 8-12

系统信息

→ **OTP writer connect to LPT1**
OTP writer 已与 LPT1 相连接

→ **Cannot connect to ICE**
OTP writer 和 HT-ICE 或 printer port 的连接发生问题。

→ **Invalid EV Chip**
OTP writer 不支持此 HT-ICE 上的 EV chip 烧录 IC。若要有正确的动作，必须更换 HT-ICE。

→ **Connect to OTP writer through ICE**
OTP writer 已通过 ICE 连接成功。

→ **Cannot find HandyWtiter，please connect it to ICE**
Or this OTP writer is an old version
HT-ICE 已接到 Print Port 上，但 OTP writer 没有接到 HT-ICE 上。也可能使用的是旧版的 OTP writer（THANDYOTP-A），所以无法侦测连线与否。若为前者，请将 OTP writer 连接到 ICE 上。

→ **File PID：ADh，OID：50h**
所打开文件记录的 power-on ID 为 ADh，Software ID 为 50h。

→ **Invalid OTP file format**
已打开文件的格式不正确。

→ **The chip PID：ADh，OID：50h doesn't match with the file PID：ADh，OID：51h**
Are you sure to continue?
OTP 单片机的类型与已打开文件所支持的类型不符合。

→ **Chip ROM size：0400h，File ROM size：0800h. System will set ROM size as 0400h.**
Are you sure to continue?
OTP 单片机可写入的空间大小为 400h，但是文件内容的大小为 800h，OTP writer 将只写入 400h 的内容到 OTP 单片机内。

→ **Addr：xxxxh，Data：yyyyh，Rdata：zzzz**
Program/Option Verify failed!!
验证 Program 或 Option 数据时发生错误。
失败原因：从 OTP 单片机中位置 xxxxh 读出的数据 zzzzh，与计算机内存内的数据 yyyyh 不一致。

→ **Addr：xxxxh，Data：zzzzh**
Not Blank!
OTP 单片机的存储器位置 xxxxh 的内容是 zzzzh，并非空的，无法执行所指定的指令。

→ **Chip mismatched!**
在 OTP writer 上的 OTP 单片机与.OTP 文件所记录的 OTP 单片机不是同一型号，无法执行所指定的命令。

→ **Chip is locked!**
在 OTP writer 上的 OTP 单片机已经被锁住。无法执行所指定的命令。

→ **No data to verify / program!**
在执行 Verify 或 Program 命令之前，必须载入.OTP 文件。请参考 HT-HandyWriter 烧录功能的 Open 命令。

第二部分

开发语言与工具

第九章

汇编语言和编译器

9

源程序由汇编语言程序构成，由盛群编译器(Holtek Assembler)编译成目标文件(Object File)，再由连接器(Linker)连接并生成任务文件(Task File)。

源程序(source program)由语句(statement)和表格(look up table)组成，在编译器进行编译或程序执行时会给予指示，而语句是由助记符(mnemonic)、操作数(operand)和注解(comment)组成。

常用符号

下表描述了文章中所用到的常用符号

范例	描述
[*optional items*]	中括号内的项目是可选择的，下列的命令行语法中： **HASM** [*options*] *filename* [;] [*options*]和[;]都是可选的，而 *filename* 则必须设定。但是在指令操作数中的中括号则是指定存储器地址之用，必须要有。
{*choice1* | *choice2*}	大括号和垂直线代表两个或更多的选项，大括号圈出这些选项而垂直线则用来分隔选项，只能有一个选项被选出。
Repeating elements…	三个连续的点表示允许输入更多同样形式的数据，例如以下的指令格式： **PUBLIC** *name1* [,*name2* [,…]] *name2* 之后的三个连续点表示允许输入更多的名字，只要每两个名字之间用逗号隔开即可。

语句语法

语句的语法格式如下：

[*名称*] [*操作项*] [*操作数项*] [*;注解*]

- 上述四个成员不一定都要指定。
- 每两个成员之间（除了注解）最少要以一个空格或一个 tab 符号分隔开。
- 成员的字型无大小写之分，换言之，编译器在编译之前会将小写字母改为大写字母。

名称

语句前可有标号以便于其它语句使用，如果名称当做标号使用，则必须在名称后紧接一个冒号（:）。名称由下列字符组成：

A～Z a～z 0～9 ？ _ @

在使用上有以下的限制：

- 不可使用数字 0～9 作为名称的第一个字符。
- ?不能单独作为名称。
- 只有最前面的 31 个字符被认可。

操作项

操作项定义两种类型的语句，伪指令与指令。伪指令用来指导编译器如何在编译时生成目标码，指令则是引导单片机执行各种运算，两者都会在编译时生成目标码，而指令被编译出的目标码会在执行时指导单片机的运作。

操作数项

操作数项定义伪指令与指令所使用的数据，由符号、常量、表达式和寄存器所组成。

注释

注释是对程序代码的一种叙述与说明，编译器不会编译它。任何在分号之后的文字均被视为注释。

汇编伪指令

汇编伪指令用来指导编译器如何在编译时生成目标码，汇编伪指令可以依其行为细分如下。

条件汇编伪指令

条件区段的格式如下：

IF
statements
[**ELSE**
statements]
ENDIF

→ **语法**

IF *expression*
IFE *expression*

- 说明

  伪指令 **IF** 和 **IFE** 对其后的 *expression* 进行检测。

  如果 *expression* 的数值为真，换言之不为零，则在 **IF** 与 **ELSE** 或 **IF** 与 **ENDIF**（没有 **ELSE**）之间所有的语句会被编译。

  如果 *expression* 的数值为假，换言之为零，则在 **IFE** 与 **ELSE** 或 **IFE** 与 **ENDIF**（没有 **ELSE**）之间所有的语句会被编译。

- 范例

```
IF   debugcase
          ACC1     equ  5
          extern   username: byte
ENDIF
```

在此范例中，如果符号 debugcase 的数值不为零，则变量 ACC1 的数值将被设定为 5 同时 username 被宣告为外部变量。

→ **语法**

IFDEF *name*
IFNDEF *name*

- 说明

  IFDEF 和 **IFNDEF** 的差异在检测 *name* 是否被定义，只要 *name* 已在前面定义为标号、变量或符号，则在 **IFDEF** 与 **ENDIF** 之间的语句都会被编译，相反的如果 *name* 还未被定义，则在 **IFNDEF** 与 **ENDIF** 之间的语句会被编译，条件汇编伪指令提供最多 7 层的嵌套。

- 范例

```
IFDEF       buf_flag
            buffer  DB  20 dup (?)
ENDIF
```

  在此范例中，只要 buf_flag 被事先定义，即配置存储器给 buffer。

文件控制伪指令

→ **语法**

INCLUDE *file-name*
或
INCLUDE *“file-name”*

- 说明

  此伪指令会在编译时，将包含入文件 *file-name* 的内容，嵌入至当前的源程序文件，并被视为源程序。编译器提供最多 7 层的嵌套。

- 范例

```
INCLUDE     macro.def
```

  在此范例中，编译器将包含文件 macro.def 内的源程序，嵌入至当前的源程序文件。

→ **语法**

PAGE *size*

- 说明

  此伪指令指定程序列表文件(program listing file)中每一页的行数，其范围介于 10 行至 255 行之间，编译器的默认值为 60 行。

- 范例

```
PAGE 57
```

  在此范例中，程序列表文件的每一页最多为 57 行。

→ 语法
.LIST
.NOLIST

- 说明

伪指令**.LIST** 和**.NOLIST** 用来决定是否要将源程序行储存到程序列表文件(program listing file)。**.NOLIST** 会停止将其后的源程序存写到程序列表文件，而**.LIST** 则会将其后的源程序行存写到程序列表文件。编译器的默认值为**.LIST**。

- 范例

```
.NOLIST
mov  a, 1
mov  b1, a
.LIST
```

上面的范例中，被**.NOLIST** 和**.LIST** 所包围的两条指令将不会被存写到程序列表文件。

→ 语法
.LISTMACRO
.NOLISTMACRO

- 说明

伪指令**.LISTMACRO** 会引导编译器将宏指令中包括注释的所有语句都存写到程序列表文件中。伪指令**.NOLISTMACRO** 则中止写入所有宏指令的语句。编译器默认值为**.NOLISTMACRO**。

→ 语法
.LISTINCLUDE
.NOLISTINCLUDE

- 说明

.LISTINCLUDE 会引导编译器将所有包含文件(included files)的内容写入程序列表文件中，**.NOLISTINCLUDE** 则会引导编译器不要将其后的包含文件的内容写进程序列表文件。编译器的默认值为**.NOLISTINCLUDE**。

→ 语法
MESSAGE *'text-string'*

- 说明

伪指令 **MESSAGE** 引导编译器将 *text-string* 显示于屏幕上，*'text-string'*的字符必须使用一对单引号括起来。

→ 语法

ERRMESSAGE *'error-string'*

- 说明

  伪指令 **ERRMESSAGE** 引导编译器显示错误信息，*'error-string'*的字符必须使用一对单引号括起来。

程序伪指令

→ 语法（注释）

; text

- 说明

  注释是以分号(semicolon)开始的字符所组成，而以回车/换行符结束。

→ 语法

name **.SECTION** [*align*] [*combine*] *'class'*

- 说明

  伪指令 **.SECTION** 标明程序段(program section)或数据段(data section)的起始地址，程序段起始地址的类型与程序段连接的方式等。程序段是由指令或指令加上数据所组成，这些指令及数据的地址则是以该程序段的段名 *name* 为起始标准而定出的。

  程序段的段名 *name* 可以是唯一的或者是与程序中其它段的段名相同。若两个程序段设定有完全相同的名称(complete name)，则被视为是同一个程序段。完全相同的名称是表示任何两个程序段的段名 *name* 及类别名 *class* 都相同。

  选项 *align* 定义程序段起始地址的类型，可以用下列中的一种：

  BYTE 以任意字节地址(byte)当做起始地址（编译器的默认形式）

  WORD 以字地址(word，两个字节,即偶数地址)当做起始地址

  PARA 以节段地址(paragraph，16 的倍数)当做起始地址

  PAGE 以分页地址(page，256 的倍数)当做起始地址

  针对 CODE 类别(class)的程序段，是以一条指令当做一个字节地址。**BYTE** 会将程序段的起始地址安排在任何指令的地址，**WORD** 则将程序段的起始地址安排在偶数的指令地址，**PARA** 将程序段的起始地址安排在 16 倍数的指令地址，而 **PAGE** 则将程序段的起始地址安排在 256 倍数的指令地址。对于 DATA 类别的程序段而言，是以一个字节（8 位/字节）当做地址的计算单位。**BYTE** 会将数据段的起始地址安排在任何字节地址，**WORD** 会将数据段的起始地址安排在偶数地址，**PARA** 则将数据段的起始地址安排在

16 倍数的地址，而 **PAGE** 会将数据段的起始地址安排在 256 倍数的地址。

选项 `combine` 定义如何结合名称相同的程序段的方法，可选用下列中的一种：

– COMMON

将具有完全相同名称的所有程序段的起始地址安排在同一个地址，所使用的存储器长度则是以最长的程序段的长度为准。

– AT *address*

此选项是指定程序段的起始地址为 address，一个固定地址。编译器及连接器不能把它安排到其它的地址，而其内的标号(label)和变量(variable)的地址可以直接从 *address* 计算出。除了不可有前置引用(forward reference)的变量或符号外，可以使用任何合乎规定的表达式来表示 address，而运算结果的数值必须是合法的 ROM/RAM 存储器地址，且不能超出 ROM/RAM 的大小范围。

如果没有设定 `combine` 的形式，则此程序段是可结合的，换句话说，此程序段和其它具有完全相同名称的程序段可以连接成一个单一的程序段。这些具有完全相同名称的程序段可以分别定义在不同的源程序文件。

`Class` 则是定义程序段的类别。相同类别的程序段被安排在存储器中的连续区域，以其输入的先后顺序一个个紧接地安排在存储器中。类别名称为 **CODE** 的程序段将会放置在程序存储器(program memory；ROM)，而类别名称为 **DATA** 的数据段则是储存在数据存储器(data memory；RAM)。在此伪指令之后，直到下一程序段伪指令之前的所有指令及数据，都属于此程序段。如果没有下一个程序段伪指令，则一直到程序文件的结尾都属于此程序段。

→ 语法

ROMBANK *banknum section-name* [,*section-name*,...]

- 说明

此伪指令是用来宣告程序存储器(program memory)的某一区块(bank)所包含的程序段。*banknum* 指定程序存储器的区块编号，范围从 0 到单片机最大的程序存储器区块数。*section-name* 则是先前已定义的程序段的名称，可以在同一个存储器区块内宣告不只一个程序段，只要这些被宣告的程序段的总和不超过 8K。如果程序中没有宣告此伪指令，则所有类别为 CODE 的程序段都被视为属于区块 0（bank 0），若某个类别为 CODE 的程序段没有被宣告为属于任何程序存储器的区块内，此程序段将被视为属于区块 0。

→ 语法

RAMBANK *banknum section-name* [,*section-name*,...]

- 说明

  此伪指令与 **ROMBANK** 相似，不同的地方是宣告数据存储器(data memory)的区块所包含的数据段(data section)。数据存储器区块的大小则为 256 字节。

→ 语法

END

- 说明

  此伪指令宣告程序的结束，因此应该避免在任何包含文件(included file)中加入此伪指令。避免编译器编译到此伪指令后就结束程序的编译流程，之后的指令及伪指令就不会被编译。

→ 语法

ORG *expression*

- 说明

  此伪指令会将 *expression* 的计算数值设定给编译器的地址计数器(location counter)，其后的程序代码和数据偏移地址将根据 *expression* 所计算的偏移量做相对的调整。程序代码和数据偏移量与伪指令 **ORG** 所在的程序段的起始地址有关，程序段的属性会决定偏移量的实际值(是绝对地址或相对地址)。

- 范例

```
ORG 8
mov A, 1
```

在此范例中，指令 `mov A, 1` 的地址是在程序段的第 8 个地址。

→ 语法

PUBLIC *name1* [,*name2* [,...]]
EXTERN *name1*:*type* [,*name2*:*type* [,...]]

- 说明

  伪指令 **PUBLIC** 用来宣告可被其它程序文件中的程序所使用的变量或标号，也就是公用变量或标号。另一方面，伪指令 **EXTERN** 则用来宣告程序将使用的外部变量、标号或符号。可使用下列四种形式中的一种：**BYTE**、**WORD**、**BIT**（这三种形式适用于数据变量）和 **NEAR**（用于调用或跳转的标号形式）。

- 范例

```
PUBLIC  start, setflag
EXTERN  tmpbuf:byte
CODE        .SECTION ‘CODE’
start:
      mov    a, 55h
      call   setflag
      ...
setflag     proc
      mov    tmpbuf, a
      ret
setflag     endp
end
```

  在此范例中，标号 start 和程序 setflag 都被宣告为公用变量，而其它源程序文件中的程序可以使用这些变量。变量 tmpbuf 则被宣告为外部变量，则需要有一个名称为 tmpbuf 的 byte 型变量定义在其它的源程序文件、而且被宣告为公用变量。

→ 语法

name **PROC**
name **ENDP**

- 说明

  伪指令 **PROC** 和 **ENDP** 用来定义一段可被其它程序调用或跳转的程序代码。必须要指定一个名称 *name* 给 **PROC** 代表此程序(procedure)第一条指令的地址，而编译器会将标号的值设定至地址计数器中。

- 范例

```
toggle      PROC
mov         tmpbuf, a
mov         a, 1
xorm        a, flag
mov         a, tmpbuf
ret
toggle      ENDP
```

→ **语法**

[*label*:] **DC** *expression1* [,*expression2* [,...]]

- 说明

  伪指令 **DC** 会将 *expression1* 及 *expression2* 等的值储存在存储器的连续位置里，此伪指令只能使用于 CODE 类别的程序段之内。*expression1* 及 *expression2* 计算的数值将视单片机的程序存储器的宽度大小而定，编译器会将任何多余的位清除掉，*expression1* 必须为数值或标号，此伪指令通常被用在程序段之内建立表格以便查询。

- 范例

```
table: DC   0128H, 025CH
```

  在此范例中，编译器会预留两个地址的 ROM 空间、并将 0128H 和 025CH 储存至这两个地址中。

数据定义伪指令

汇编程序由语句和注释组成，语句或注释则是由字符、数字和名称构成。汇编语言提供整数数字，整数可由二进制、八进制、十进制或十六进制加以表示(配合字尾的基数)，如果未选基数，则编译器会使用默认值(十进制)，下表为可用的基数。

基底	类型	数字
B	二进制	01
O	八进制	01234567
D	十进制	0123456789
H	十六进制	0123456789ABCDEF

→ **语法**

```
[name]  DB  value1 [,value2 [,...]]
[name]  DW  value1 [,value2 [,...]]
[name]  DBIT
[name]  DB  repeated-count DUP(?)
[name]  DW  repeated-count DUP(?)
```

- 说明

  上述伪指令会引导编译器在数据存储器(data memory)内保留空间给变量 *name* (如果有指定 *name*)。存储器保留的空间大小则由其后的个数及数据类型，或由重复次数及数据类型来决定。由于单片机的数据存储器无法事先记录数据内容，编译器不会对数据存储器做初始值的设定，因此 *value1* 和 *value2* 必须为“？”表示只是保留存储器空间给程序执行时使用，并没有设定其初始值。**DBIT** 只保存一个位，编译器会将每 8 个 **DBIT** 整合在一起并且保留一个字节给这 8 个 **DBIT** 变量。

- 范例

```
DATA        .SECTION     'DATA'
tbuf        DB  ?
chksum      DW  ?
flag1       DBIT
sbuf        DB  ?
cflag       DBIT
```

  在此范例中，编译器保留字节地址 0 给变量 tbuf、字节地址 1 与 2 给变量 chksum、字节地址 3 的第 0 位给变量 flag1、字节地址 4 给变量 sbuf 以及字节地址 3 的第 1 位给变量 cflag。

→ **语法**

name **LABEL BIT | BYTE | WORD**

- 说明

  此伪指令会将 *name* 的地址设定为与其后的变量相同的存储器地址。

- 范例

```
lab1        LABEL        WORD
d1          DB  ?
d2          DB  ?
```

  在这个范例中，d1 是 lab1 的低字节，而 d2 则是 lab1 的高字节。

→ **语法**

name **EQU** *expression*

- 说明

  通过给 *name* 指定 *expression*，**EQU** 伪指令会生成一个新的数值符号、别名或文字符号(*name*)来代表 *expression*。数值符号是一个代表 16 位值的名称、别名则是另一个符号的名称、而文字符号则是代表一串字符组合的名称。*name* 必须是唯一的，即先前未被定义过。*expression* 可以是一个整数、字符串常量、指令助记符、数学表达式或地址表达式。

- 范例

```
accreg        EQU    5
bmove         EQU    mov
```

  在这个范例中，变量 accreg 等于 5，而 bmove 相当于指令 mov。

宏指令

宏指令用来定义一个名称来代表一段源程序语句(指令及伪指令)，而在源程序文件中可以重复用这个名字以取代这段语句，也就是程序中所有用到此段语句的地方都可用这个名字代替。在编译时，编译器会自动将每一个宏指令的名称用宏指令所定义的程序语句来取代。

宏指令可以在源文件的任何地方被定义，只要调用此宏指令是在宏指令定义之后即可。宏指令的定义中，可以调用先前已经被定义的其它宏指令，如此将形成一种嵌套的结构，编译器提供最多 7 层的嵌套。

→ **语法**

name **MACRO** [*dummy-parameter* [,...]]
    statements
    ENDM

在宏指令中，可以使用伪指令 **LOCAL** 来定义只能在宏指令本体内使用的变量。

→ **语法**

name **LOCAL** *dummy-name* [,...]

- 说明

宏指令 **LOCAL** 用来定义只能在宏指令本体内使用的符号，使用时必须定义在 **MACRO** 伪指令之后的第一行。*dummy-name* 是一个暂时使用的名称，当宏指令被调用展开时，它将被一个唯一的名称所取代，编译器会对 *dummy-name* 生成对应的实际名称，这个实际名称的格式为??digit，其中 digit 数字为十六进制且范围由 0000 至 FFFF。当 **MACRO/ENDM** 的定义区段中使用到标号(label)时，最好将此标号加入 **LOCAL** 伪指令中，否则当 **MACRO** 被源文件多次调用时，相同的标号名称会重复出现在程序中,编译器将会发布程序错误的信息。

下面的范例中，tmp1 和 tmp2 都是虚参数，当调用此宏指令时，都会被实际参数所取代。label1 和 label2 都被宣告为 **LOCAL**，如果没有其它的 **MACRO** 被调用，在第一次调用时将分别被??0000 和??0001 所取代，如果没有宣告 **LOCAL**，label1 和 label2 则会类似于源程序中的标号宣告，而在第二次调用此宏指令时，就会出现重复定义的错误信息。

```
Delay  MACRO   tmp1, tmp2
LOCAL          label1, label2
      mov      a, 70h
      mov      tmp1, a
label1:
      mov      tmp2, a
label2:
      clr      wdt1
      clr      wdt2
      sdz      tmp2
      jmp      label2
      sdz      tmp1
      jmp      label1
      ENDM
```

下面的源程序将会调用名为 Delay 的宏指令

```
; T.ASM
; Sample program for MACRO.
.ListMacro
Delay MACRO  tmp1, tmp2
      LOCAL  label1, label2
      mov    a, 70h
      mov    tmp1, a
label1:
      mov    tmp2, a
label2:
      clr    wdt1
      clr    wdt2
      sdz    tmp2
      jmp    label2
      sdz    tmp1
      jmp    label1
      ENDM

data .section ‘data’
BCnt db ?
SCnt db ?

code .section at 0   ‘code’
Delay BCnt, SCnt
end
```

编译器会将宏指令 Delay 展开如下列的程序，请注意在宏指令本体内的第 4 行到第 17 行，它们的位置偏移量(offset)都是 0000，也就是宏指令在定义时，本体内的指令并不占用存储器空间。在程序第 24 行调用 Delay 宏指令时，它就被展开成 11 行并且排放在宏指令之后，这 11 行的指令则沿用相同的行数(line number)第 24 行。同时，虚参数 `tmp1` 和 `tmp2` 分别被实际参数 `BCnt` 和 `SCnt` 所取代。

```
File: T.asm            Holtek Cross-Assembler    Version 2.80     Page 1
 1  0000                    ; T.ASM
 2  0000                    ; Sample Program for MACRO.
 3  0000                    .ListMacro
 4  0000                    Delay MACRO  tmp1, tmp2
 5  0000                          LOCAL  label1, label2
 6  0000                          mov    a, 70h
 7  0000                          mov    tmp1, a
 8  0000                    label1:
 9  0000                          mov    tmp2, a
10  0000                    label2:
11  0000                          clr    wdt1
12  0000                          clr    wdt2
13  0000                          sdz    tmp2
14  0000                          jmp    label2
15  0000                          sdz    tmp1
16  0000                          jmp    label1
17  0000                          ENDM
18  0000
19  0000                    data .section 'data'
20  0000  00                BCnt db ?
21  0001  00                SCnt db ?
22  0002
23  0000                    code .section at 0 'code'
24  0000                    Delay BCnt, SCnt
24  0000  0F70          1         mov    a, 70h
24  0001  0080         R1         mov    BCnt, a
24  0002                1    ??0000:
24  0002  0080         R1         mov    SCnt, a
24  0003                1    ??0001:
24  0003  0001          1         clr    wdt1
24  0004  0005          1         clr    wdt2
24  0005  1780         R1         sdz    SCnt
24  0006  2803          1         jmp    ??0001
24  0007  1780         R1         sdz    BCnt
24  0008  2802          1         jmp    ??0000
25  0009                    end

      0 Errors
```

汇编语言指令

指令的语法形式如下：

[*name*:] mnemonic [*operand1* [,*operand2*]] [; *comment*]

其中

name:	→ 符号名称
mnemonic	→ 指令名称（关键字）
operand1	→ 寄存器 存储器地址
operand2	→ 寄存器 存储器地址 立即数

名称

名称是由字母、数字或特殊字符所组成的，可以当标号(label)使用。当标号使用时，必须在名称后面紧接一个冒号(colon)。

助记符

助记符是源程序中使用的指令名称。

操作数、运算子和表达式

操作数（源操作数或目的操作数）定义被指令所使用的数值，它们可以是常量、变量、寄存器、表达式或关键字。当使用指令时，必须谨慎选择正确的操作数，即源操作数和目的操作数。钱字符号 $ 是一个特殊的操作数，它代表当前的地址。

表达式是由操作数所组成，在程序编译时用来计算出数值或存储器地址。表达式是常量、符号以及任何被算术运算子分隔的常量和符号组合。

运算子定义表达式中各操作数之间的运算动作，编译器提供了许多运算子去处理操作数，有些运算子只处理常量，有些则处理存储器数值，也有两者兼具的。如果运算子只是处理常量,则在程序编译时就会直接计算出数值。以下是编译器所提供的运算子。

- 算术运算子　+ – * / %（MOD）
- SHL 和 SHR 运算子
  – 语法
  expression **SHR** *count*
  expression **SHL** *count*
  这些位平移运算子的值全都为常量，*expression* 依照 *count* 所指定的位数目向右移**(SHR)**或向左移**(SHL)**，如果被平移的位超过有效位数时，则对应的位会以 0 填满，如：

```
mov  A, 01110111b  SHR 3  ; result  ACC=00001110b
mov  A, 01110111b  SHL 4  ; result  ACC=01110000b
```

- 逻辑运算子 NOT、AND、OR、XOR
  – 语法
  NOT *expression*
  expression1 **AND** *expression2*
  expression1 **OR** *expression2*
  expression1 **XOR** *expression2*
  NOT　各位的 1 阶补码
  AND　各位 AND 运算
  OR　各位 OR 运算
  XOR　各位 XOR 运算
- 偏移运算子
  – 语法
  OFFSET *expression*
  OFFSET 运算子会生成 *expression* 的偏移地址，*expression* 可以是标号、变量或其它直接存储器地址的操作数，被 **OFFSET** 运算子所返回的数值必须是立即数。
- LOW、MID 和 HIGH 运算子
  – 语法
  LOW *expression*
  MID *expression*
  HIGH *expression*
  如果 *expression* 的结果是一个立即数的话，则 **LOW/MID/HIGH** 运算子会生成 *expression* 的值，而且是分别取此数值的低/中/高字节。但是如果 *expression* 是标号，则 **LOW/MID/HIGH** 运算子将取得此标号所在的程序存储器地址的低/中/高字节的数值。

- BANK 存储区块运算子
  – 语法
    BANK *name*

  BANK 运算子会生成程序段所在的存储器区块的编号，此程序段是由 *name* 所宣告的。如果 *name* 是标号类型，生成的则是 ROM 程序存储器区块，如果 *name* 为数据变量则区块生成 RAM 数据存储器区块。存储器的数值格式与寄存器 BP 的格式相同，请参考各单片机的规格，不同的单片机可能有不同的 BP 格式。

  范例 1:

```
mov  A, BANK start
mov  BP, A
jmp  start
```

  范例 2:

```
mov  A, BANK var
mov  BP, A
mov  A, OFFSET var
mov  MP1, A
mov  A, R1
```

- 运算子的优先权

优先权	运算子
1(Highest)	( ), [ ]
2	+ , – (unary), LOW, MID, HIGH, OFFSET, BANK
3	*, /, %, SHL, SHR
4	+, – (binary)
5	> (greater than), >= (greater than or equal to), < (less than), <= (less than or equal to)
6	= = (equal to), != (not equal to)
7	! (bitwise NOT)
8	& (bitwise AND)
9(Lowest)	| (bitwise OR), ^ (bitwise XOR)

其它

前置引用

当标号、变量名称和其它符号在源代码中被宣告之前，编译器允许它们被使用(前置命名引用)，但是在伪指令 **EQU** 右边的符号是不允许前置引用的。

局部标号

局部标号有固定形式的，如 $number。其中 number 可以为 0 至 29，局部标号除了可以重复使用外，其它功用与一般标号是相同的。局部标号必须使用在任意两个连续的标号之间而且同样的局部标号名称也可以用在其它的两个连续标号之间。在编译源程序文件之前，编译器会将每一个局部标号转换成唯一的标号。任何两个连续标号之间，最多可以定义 30 个局部标号。

范例

```
Label1:                         ; label1
          $1:                   ;; local label
                mov a, 1
                jmp $3
          $2:                   ;; local label
                mov a, 2
                jmp $1
          $3:                   ;; local label
                jmp $2
Label2:                         ; label
                Jmp $1
          $0:                   ;; local label
                jmp Label1
          $1:   jmp $0
Label3:
```

汇编语言保留字

下表是汇编语言上使用的保留字。

- 保留字（伪指令、运算子）

$	DUP	INCLUDE	NOT
*	DW	LABEL	OFFSET
+	ELSE	.LIST	OR
-	END	.LISTINCLUDE	ORG
.	ENDIF	.LISTMACRO	PAGE
/	ENDM	LOCAL	PARA
=	ENDP	LOW	PROC
?	EQU	MACRO	PUBLIC
[ ]	ERRMESSAGE	MESSAGE	RAMBANK
AND	EXTERN	MID	ROMBANK
BANK	HIGH	MOD	.SECTION
BYTE	IF	NEAR	SHL
DB	IFDEF	.NOLIST	SHR
DBIT	IFE	.NOLISTINCLUDE	WORD
DC	IFNDEF	.NOLISTMACRO	XOR

- 保留字（指令助记符）

ADC	HALT	RLCA	SUB
ADCM	INC	RR	SUBM
ADD	INCA	RRA	SWAP
ADDM	JMP	RRC	SWAPA
AND	MOV	RRCA	SZ
ANDM	NOP	SBC	SZA
CALL	OR	SBCM	TABRDC
CLR	ORM	SDZ	TABRDL
CPL	RET	SDZA	XOR
CPLA	RETI	SET	XORM
DAA	RL	SIZ	
DEC	RLA	SIZA	
DECA	RLC	SNZ	

- 保留字（寄存器名称）

A	WDT	WDT1	WDT2

编译器选项

编译器选项可以通过 HT-IDE3000 中的 Options 菜单的 Project 命令来设定，编译器的选项位于 Project Option 对话框的中心部分，可在符号定义(Define Symbol)编辑框中定义符号。

→ **语法**

```
symbol1 [=value1] [,symbol2 [=value2 ] [,...]]
```

- 范例

```
debugflag=1, newver=3
```

生成列表文件的检查框可用来指定是否要生成列表文件(Listing file)，如果检查验框被选中，则要生成列表文件，否则将不会生成列表文件。

编译列表文件格式

编译列表文件包含源程序的列表和概要信息，每页的第一行是标题，内容则包括公司名称、编译器版本、源文件名称、编译的日期，时间以及页码。

源程序列表

在源程序中的每行语句都以下列的格式输出到编译列表文件：

```
[line-number] offset [code] statement
```

- *Line-number* 是指语句在源程序文件的第几行,从第一个语句开始计算起(4 位十进制数)。
- *offset*-是从语句所在的程序段开始到这个语句的存储器地址的偏移量(4 位十六进制数)。

- `code`-只有会生成机器代码(machine code)或数据的语句才会出现此项（两个 4 位十六进制数）。
  如果数值在编译时已确定的话，会用十六进制数字表示 `code` 的数值，否则的话，将使用适当的标志位表明应该使用何种方式去计算此数值。下列两个标志位可能会出现于 `code` 项目之后。
  R → 需要重新安置地址 (连接器解决此状况)
  E → 需要参考外部符号 (连接器解决此状况)
  下列标志位可能会出现于 `code` 项目之前。
  = → **EQU** 或等号
  `code` 项目中可能出现下列的符号或数字。
  ---- → 代表程序段的起始地址 (连接器会解决此符号)
  nn[xx]→ **DUP** 符号：nn **DUP**(?)重复次数
- `statement`-与源程序文件中相同的语句或是宏指令所展开的语句。在语句之前可能会出现下列的符号。
  n → 宏指令展开时的嵌套层次
  C → 此语句是从包含文件(**INCLUDE** 文件) 引进的
- 总结

```
0         1         2         3         4         5         6
1234567890123456789012345678901234567890123456789012345678901234567890...
IIII  oooo  hhhh  hhhh  EC  source-program-statement
                        Rn
```

IIII → 行号（4 位数，向右靠齐）
oooo → 机器代码的地址偏移量（4 位数）
hhhh → 两个 4-位数的机器代码
E → 参考外部数据
C → 从包含文件加入的语句
R → 需要重新安置地址
n → 宏指令展开后的嵌套层次

编译总结

在编译列表文件的结尾处会统计此次编译所发生的警告及错误的总数。

其它

在编译期间如果发生错误，则错误信息和编号会直接出现在发生错误的语句下方。

→ 编译列表文件的范例

```
File: SAMPLE.ASM   Holtek Cross-Assembler  Version 2.86      Page 1

  1  0000                 page 60
  2  0000                 message   ‘Sample Program 1’
  3  0000
  4  0000                 .listinclude
  5  0000                 .listmacro
  6  0000
  7  0000                 #include “sample.inc”

  1  0000              C  pa    equ    [12h]
  2  0000              C  pac   equ    [13h]
  3  0000              C  pb    equ    [14h]
  4  0000              C  pbc   equ    [15h]
  5  0000              C  pc    equ    [16h]
  6  0000              C  pcc   equ    [17h]
  7  0000              C

  8  0000
  9  0000                 extern extlab : near
 10  0000                 extern extb1 : byte
 11  0000
 12  0000                 clrpb macro
 13  0000                 clr pb
 14  0000                 endm
 15  0000
 16  0000                 clrpa macro
 17  0000                 mov a, 00h
 18  0000                 mov pa, a
 19  0000                 clrpb
 20  0000                 endm
 21  0000
 22  0000                 data .section ‘data’
 23  0000  00             b1    db ?
 24  0001  00             b2    db ?
 25  0002  00             bit1  bit
 26  0003
 27  0000                 code .section ‘code’
 28  0000  0F55           mov  a, 055h
 29  0001  0080        R  mov  b1, a
 30  0002  0080        E  mov  extb1, a
 31  0003  0FAA           mov  a, 0aah
 32  0004  0093           mov  pac, a
 33  0005                 clrpa
 33  0005  0F00        1 mov  a, 00h
 33  0006  0092        1 mov  [12h], a
 33  0007              1 clrpb
 33  0007  1F14        2 clr  [14h]
 34  0008  0700        R  mov  a, b1
 35  0009  0F00        E  mov  a, bank extlab
 36  000A  0F00        E  mov  a, offset extb1
 37  000B  2800        E  jmp  extlab
 38  000C
 39  000C  1234 5678      dw 1234h, 5678h, 0abcdh, 0ef12h
           ABCD EF12
 40  0010                 end

        0 Errors
```

第十章

盛群 C 语言

10

简介

盛群半导体公司的 C 编译器基本上是建构于 ANSI C，由于受限于盛群单片机的硬件结构，因此只能支持部分的 ANSI C，本章节主要用来说明盛群 C 编译器所提供的 C 程序语言。

本章节包含以下的主题：

- C 语言的程序结构
- 标识符
- 数据类型
- 常量
- 运算符
- 程序流程控制
- 函数
- 指针与数组
- 结构体与共用体
- 前置处理程序伪指令
- 盛群 C 语言的扩充功能与限制

C 语言的程序结构

C 语言程序由语句、注释和前置处理程序伪指令组合而成。

语句

语句由变量、常量、运算符和函数共同组成，以分号作为结束符，并且可以执行以下的动作：

- 宣告数据变量与数据结构
- 定义数据空间
- 执行数学与逻辑运算
- 执行程序的控制动作

一行程序可以包含多个语句，复合语句由一个或多个被包含在一对大括号内的语句组成，并且可将其当成单一的语句来使用。有些语句和前置处理程序伪指令必须在盛群 C 源程序文件中使用。以下是一个整体轮廓的例子：

```
void main ()
{
/* user application source code */
}
```

源程序中必须要定义主函数 main。项目中可能会包含不只一个源程序文件，但只有一个源程序文件中可以定义主函数 main。

注释

注释经常用于在文件中解释源程序语句的意义与作用，除了在 C 关键字的中间、函数名称之间或变量名称之间外，注释可以放置在程序中的任何位置。C 编译器不对注释做处理。注释不可嵌套(nest)。盛群 C 编译器提供两种注释方式，块注释与行注释。

→ 块注释

块注释开始于/*而结束于*/，以下为一范例：

```
/* this is a block comment */
```

块注释的结束符*/也许会与块注释的开始符/*位于不同的程序行，介于开始符/*和结束符*/ 之间所有的文字或符号都会被 C 编译器当成注释而不予编译。

→ 行注释

行注释开始于//直到此行的结束为止。在双斜线之后的文字或符号均视为注释。范例如下：

```
// this is a line comment
```

标识符

标识符的名称包含连续的字母、数字或下划线，不过需要遵守下列规则：

- 第一个字符不可为数字
- 最长只能有 31 个字符
- 大写字母与小写字母是不同的
- 不可以使用保留字

保留字

下列为盛群 C 编译器所提供的保留字，注意要小写。

auto	bit	break	case	char
const	continue	default	do	else
enum	extern	for	goto	if
int	long	return	short	signed
static	struct	switch	typedef	union
unsigned	void	volatile	while	

盛群 C 编译器不提供 **double**、**float** 和 **register** 这三个保留字。

数据类型

数据类型与大小

盛群 C 编译器提供四种基本数据类型，分别为：

bit	单一的位
char	占用一个字节的字符
int	占用一个字节的整数
void	数值的空集合，用于函数没有返回值的类型。

接下来为可使用的限定词，分别为：

限定词	适用的数据类型	作用
const	any	将数据放入 ROM 地址区
long	int	生成一 16 位的整数
short	int	生成一 8 位的整数
signed	char, int	生成一个有符号的变量
unsigned	char, int	生成一个无符号的变量

下列为数据类型、大小与范围，分别为：

数据类型	大小	范　围
bit	1	0, 1
char	8	-128~127
unsigned char	8	0~255
int	8	-128~127
unsigned	8	0~255
short int	8	-128~127
unsigned short int	8	0~255
long	16	-32768~32767
unsigned long	16	0~65535

宣告

在定义变量的大小及数据类型之前必须先要宣告此变量的存在。宣告的语法如下：

```
data_type variable_name [,variable_name...];
```

在该范例中，*data_type* 是合法的数据类型而 *variable_name* 是变量的名称。在函数中所宣告的变量只是此函数私有的（或局部的）变量，其它函数不可以直接存取此变量。只有当函数被调用时，此函数中的局部变量才存在及有效，当执行流程从函数回到调用的程序时，局部变量便不再有效。如果变量在所有函数之外宣告，则此变量为全局变量，即所有函数均可使用、存取此变量。

限定词 **const** 可以使用在任何变量的宣告，主要是定义此变量的值为不可改变的,也就是宣告时用 **const** 限定的这个变量会存放在 ROM 地址区。限定词 **const** 也可以使用在数组变量中，**const** 变量必须在宣告时以等号和表达式设定初始值，其它的变量在宣告时不能设定初始值。

可以利用@符号宣示变量放置在某个特定的数据存储器地址，其语法如下：

```
data_type variable_name @ memory_location;
```

上例中,*memory_location* 是指定给变量的地址。如果单片机拥有多个 RAM 存储器区块，若变量要放置于编号为 0 的 RAM 存储器区块之外时，可以利用 *memory_location* 的高字节去指定所要存放的存储器区块编号。使用者可查阅盛群单片机的规格以取得可使用的 RAM 空间信息。

范例：

```
int v1 @ 0×40;   // declare v1 in the RAM bank 0 offset 0×40
int v2 @ 0×160;  // declare v2 in the RAM bank 1 offset 0×60
```

数组也可以被宣告在特定地址：

```
int port [8] @ 0×20;    // array port takes memory location
                        // 0×20 through 0×27
```

所有被盛群 C 编译器实现的变量,除了被宣告为外部变量之外,都为静态变量。无论是静态变量或是外部变量，盛群 C 编译器都不会为其预设初始值。

注意: 变量被宣告为无符号的数据类型比宣告为有符号的数据类型能够编译出效率更高的程序代码

常量

常量可以是任何数字、单一字符或字符串。

整型常量

整型常量为 int 型数据，长常量通常以 l 或 L 结尾，无符号常量则以 u 或 U 结尾，而字尾为 ul 或 UL 则表示为无符号长常量。整型常量的数值可以用下列的形式指定：

二进制常量：以 0b 或 0B 为首的数字

八进制常量：以 0 为首的数字

十六进制常量：以 0x 或 0X 为首的数字

十进制常量：非以上为首的数字

字符型常量

字符型常量是整数，它是用单引号括起来的一个字符。字符型常量的数值就是机器字符集中的字符数值。ANSI C 把转义字符(escape sequence)当作字符型常量处理。

转义字符	说明	十六进制数值
\a	警报（铃声）字符	07
\b	退格字符	08
\f	换页字符	0C
\n	换行字符	0A
\r	回车字符	0D
\t	横向跳格字符	09
\v	竖向跳格字符	0B
\\	反斜杠字符	5C
\?	问号字符	3F
\′	单引号字符	27
\″	双引号字符	22

字符串常量

字符串常量是由一对双引号括起来的零个或多个字符（包括 ANSI C 转义字符）。字符串常量是一个字符数组并且在字符的最后附加一个隐含的零值。因此，所需要的储存空间大小是双引号括起来的字符总数再加上 1。

枚举常量

整型常量的另一种命名方法称之为枚举常量，例如：

```
enum {PORTA, PORTB, PORTC};
```

定义三个整型常量的枚举常量，并且分别分配数值。

枚举常量是 int 型（-128~127），而且也可以指定一个明确的整数值给各枚举常量，例如：

```
enum {BIG=10, SMALL=20};
```

如果没有对枚举常量指定明确的数值时，第一个枚举常量值为 0，之后的枚举常量将依序加 1。枚举语句也可以被命名，例如：

```
enum boolearn {NO, YES};
```

在枚举语句中第一个名称（NO）的值为 0，下一个的名称的值是 1。

运算符

表达式是由一串运算符及操作数所组成并且指明其运算的式子，它会遵循代数的规则以计算出数值或某些负效果。表达式中某些部分计算时的顺序将会根据运算符的执行优先权和运算符所属的群组来决定。数学上常使用的运算符的结合性及交换性规则，只能应用在具有结合性和交换性的运算符。接下来讨论各种类型的运算符。

算术运算符

共有五种算术运算符。

+	加法运算符
-	减法运算符
*	乘法运算符
/	除法运算符
%	模运算符（余数为小于除数的正数或零）

模运算符%只能使用在整数的数据类型。

关系运算符

关系运算符比较两个数值，然后根据比较结果返回 TURE（真）或 FALSE（假）。

>	大于
>=	大于或等于
<	小于
<=	小于或等于

等式运算符

等式运算符类似于关系运算符。

= =	等于
!=	不等于

逻辑运算符

逻辑运算符提供 AND、OR、和 NOT 的逻辑运算并且生成 TURE（真）或 FALSE（假）值。由&&和｜｜连接的表达式由左到右计算，只要结果生成就停止计算。如果关系表达式或逻辑表达式的结果为真(TRUE)，则表达式的结果数值为 1，否则为 0。否定运算符!用来将 0 变为 1 及 1 变为 0。

&&	逻辑 AND
｜｜	逻辑 OR
!	逻辑 NOT

位运算符

提供六种运算符用于位对位的运算。位移运算符>>和<<会对运算符左边的操作数执行向右或向左的位移动，移动的位数由运算符右边的操作数指定。单操作数运算符~生成整数的 1 阶补码(one’s complement)，也就是将 1 改为 0，将 0 改为 1。

&	位 AND
｜	位 OR
^	位 XOR
~	取 1’s 补码（位反向）
>>	右移
<<	左移

复合赋值运算符

表达式的语句中总共有 10 种复合赋值运算符。对于单纯的赋值运算就是使用一个等号，以表达式计算出的数值代表等号左边的变量。另外还提供一种直接对变量本身做运算以达到修改变量的快捷方式。

<var>	+=<expr>	变量加上 expr 的值，将结果存回变量
<var>	-=<expr>	变量减去 expr 的值，将结果存回变量
<var>	*=<expr>	变量乘以 expr 的值，将结果存回变量
<var>	/=<expr>	变量除以 expr 的值，将商数存回变量
<var>	%=<expr>	变量除以 expr 的值，将余数存回变量
<var>	&=<expr>	变量与 expr 的值做位 AND 后，将结果存回变量
<var>	｜=<expr>	变量与 expr 的值做位 OR 后，将结果存回变量
<var>	^=<expr>	变量与 expr 的值做位 XOR 后，将结果存回变量
<var>	>>=<expr>	变量向右移 expr 个位后，将结果存回变量
<var>	<<=<expr>	变量向左移 expr 个位后，将结果存回变量

递增和递减运算符

递增和递减运算符可以使用在语句本身或将其插入有其它运算符的语句中。运算符的位置表示递增和递减是要在语句的计算结果之前（前缀运算符）或是之后（后缀运算符)。

++<var> 变量先加 1，再做运算
<var>++ 运算之后，变量再加 1
--<var> 变量先减 1，再做运算
<var>-- 运算之后，变量再减 1

条件运算符

条件运算符?：是一个简洁的语句，它根据表达式的结果再去执行两个语句中的一个。

<*expr*> ? <*statement1*> : <*statement2*>

如果<*expr*>的计算结果为一非零值（真）则<*statement1*>被执行，反之（假）则执行<*statement2*>。

逗号运算符

一组用逗号分隔的表达式，由左计算到右，而左边表达式的值会被舍弃。左边表达式的结果会先行计算出并会影响右边表达式执行的结果。整个表达式执行结果的数值和数据类型将是最右边表达式的结果数值及数据类型。

范例：

```
f (a, (t=3, t+2), c);
```

上式有三个参数，而第二个参数值为 5。

运算符的优先权与结合性

下表为运算符的优先权与结合性，优先权顺序是由高到低排列，而在同一格中的运算符拥有同等的优先权，单操作数运算符（unary operator）和复合赋值运算符的结合性为从右到左，而其它运算符的结合性为从左到右。

运算符	说明	结合性
[ ]	数组元素	由左到右
( )	小括号	
→	结构体指针	
.	结构体成员	
sizeof	数据类型的长度	
++	加 1	由右到左
--	减 1	
~	取 1 阶补码	
!	逻辑非	
-	负号	
+	正号	
&	变量地址	
*	存取指针所指地址的内容	
*	乘法运算	由左到右
/	除法运算	
%	模运算	
+	加法运算	由左到右
-	减法运算	
<<	左移运算	由左到右
>>	右移运算	
<	小于	由左到右
<=	小于或等于	
>	大于	
>=	大于或等于	
==	等于	由左到右
!=	不等于	
&	按位 AND	
^	按位 XOR	
｜	按位 OR	
&&	逻辑 AND	
｜｜	逻辑 OR	
?：	条件运算	

运算符	说明	结合性
=	赋值	由右到左
*=	相乘后存入变量	
/=	相除后存入变量	
%=	取模后存入变量	
+=	相加后存入变量	
-=	相减后存入变量	
<<=	左移后存入变量	
>>=	右移后存入变量	
&=	按位 AND 后存入变量	
｜=	按位 OR 后存入变量	
^=	按位 XOR 后存入变量	
,	逗号	由左到右

类型转换

对于数据类型转换的规则而言，大都是将较小的操作数转换为较大的操作数而不致遗漏数据，例如将整数类型转换为长整数类型。从 char 转到 long 则会做正负符号的延伸。使用 cast 运算符可以将任何表达式的结果做明确的数据类型转换。例如：

```
(type-name) expression
```

`expression` 的结果将被转换为 `type-name` 所指定的数据类型。

程序流程控制

本节的语句都是用来控制程序执行的流程。同时也叙述如何使用控制语句中的关系与逻辑运算符以及如何执行循环。

→ **if-else 语句**

- 语法

```
if (expression)
statement1;
[else
statement2;
]
```

- 说明

if-else 是一种条件语句，语句区段的执行与否完全看 *expression* 的结果，如果 *expression* 的结果为非零值，则与其相关联的语句区段被执行，否则如果 **else** 的区段存在的话，与 **else** 相关联的语句区段就会被执行。**else** 语句与其关联的语句区段并不一定要存在。

- 范例

```
if   (word_count> 80)
{
     word_count=1;
     line++;
}
else
     word_count++;
```

→ **for 语句**

- 语法

```
for (initial-expression; condition-expression;
update-expression) statement;
```

- 说明

initial-expression 最先被执行且只执行一次，通常用来给循环的计数变量指定初始值，此变量必须在 **for** 循环之前被宣告。*condition-expression* 要在每一个循环执行前先计算，如果结果为一非零值则循环中的语句被执行，否则会跳出循环且在循环后的第一个语句将会是下一个被执行的语句。*update-expression* 会在循环内的语句执行完之后才被执行。**for** 语句可用来重复执行一行语句或一段语句。

- 范例

```
for  (i=0; i<10;i++)
     a[i]=b[i]; // copy elements from an array to another array
```

→ **while 语句**

- 语法

```
while (condition-expression)
      statement;
```

- 说明

while 语句是另一种形式的循环。当 *condition-expression* 不为零则 **while** 循环会执行 *statement*。在执行 *statement* 之前会先行查验 *condition-expression* 是否符合条件。

- 范例

```
i= 0;
while (b[i]!=0)
{
      a [i]=b[i];
      i++;
}
```

→ **do-while 语句**

- 语法

```
do
      statement;
while (condition-expression);
```

- 说明

do-while 语句是另一种形式的 **while** 循环。*statement* 会在 *condition-expression* 被计算之前先执行一次，因此在查验 *condition-expression* 之前至少会执行一次 *statement*。

- 范例

```
i=0;
do
{
      a [i]= b [i];
      i++;
} while (i<10);
```

→ **break 和 continue 语句**

- 语法

  break;
  continue;

- 说明

  break 语句用来强迫程序立即由 **while**、**for**、**do-whil**e 循环和 **switch** 中跳出。**break** 语句会跳过正常的结束流程，如果它发生在嵌套循环的内部，则会返回上一层的嵌套。

  Continue 语句会指示程序跳跃至循环的结束而重新开始下一轮循环。在 **while** 和 **do-while** 循环中，**Continue** 语句会强迫立即执行 *condition-expression*，而在 **for** 循环中，则会回去执行 *update-expression*。

- 范例

```
char a[10], b[10], i, j;
for (i=j=0;i<10;i++)//copy data from b[ ] to a[ ],skip blanks
{
      if (b[i]== 0) break;
      if (b[i]== 0×20) continue;
      a[j++]= b[i];
}
```

→ **goto 语句和语句标号**

- 语法

  goto *label*;

- 说明

  语句标号与变量名称的形式一样，但是其后要接冒号，其范围在整个函数中有效。

- 范例

  参考 **switch** 语句的范例。

→ **switch 语句**

- 语法

```
switch (variable)
{
        case constant1:
                statement1;
                break;
        case constant2:
                statement2;
                goto Label1;
        case constant3:
                statement3;
                break;
        default:
                statement;
Label1:  statement4;
                break;
}
```

- 说明

switch 语句的 *variable* 变量用来测试变量与列表中的常量是否吻合，当吻合时此常量所属的语句被执行，并且一直执行到遇上 **break** 语句才会停止。如果 **break** 语句不存在，则程序会执行到 **switch** 程序段的结束为止。如果没有符合的常量，则执行 **default** 所属的语句，此语句并非必要的。

if-else 语句可以用来做二选一的选择，但是当有很多选择存在时就变得很麻烦了。**switch** 语句可以做多种方式的选择，当表达式的结果符合这些选择中的一个时，就跳到相关的语句执行。它相当于多个 **if-else** 语句。**switch** 语句的限制为 **switch** 变量的数据类型必须为整数，而且只能与常量值做比较。

- 范例

```
for (i=j=0;i<10;i++)
{
        switch (b[i])
        {
                case 0: goto outloop;
                case 0×20: break;
                default:
                   a[j]=b[i];
                   j++;
                   break;
        }
}
outloop:
```

函数

在 C 语言中,所有的执行语句都必须存在于函数之内。在使用或调用函数之前,必须要定义或是宣告函数，否则 C 编译器会发出警告信息。在宣告或定义函数时，可使用两种语法，即古典形式与现代形式。针对具有多个程序存储器区块(bank)的单片机撰写程序时，函数就与变量有所不同，使用者不需要而且也没有办法将函数指定在存储器的固定区块(bank)。连接器(Linker)会将函数安排在程序存储器 ROM 的适当区块。

古典形式

```
return-type function-name (arg1, arg2, ...)
var-type arg1;
var-type arg2;
```

现代形式

```
return-type function-name (var-type arg1, var-type arg2, ...)
```

在上述两种形式中，*return-type* 是函数返回值的数据类型，如果函数没有返回值，必须将 *return-type* 宣告为 **void** 类型。*function-name* 是函数的名称,对其他所有的函数而言,它相当于一个全局变量。参数 *arg1*, *arg2* 是在此函数内使用的变量，必须指定它们的数据类型，当调用函数时，这些变量会代替形式参数储存对应的输入值。

→ **函数的宣告**

```
//  classic form
    return-type function-name (arg1, arg2, ...);
//  modern form
    return-type function-name (var-type arg1, var-type arg2, ...);
```

→ **函数的定义**

```
// classic form
return-type function-name (arg1, arg2, ...)
var-type arg1;
var-type arg2;
{
     statements;
}
// modern form
return-type function-name (var-type arg1, var-type arg2, ...)
{
     statements;
}
```

→ **函数参数的传递**

有两种函数参数传递的方法：

- 传值

  此方法是将参数值复制到函数中对应的形式参数。在函数中对形式参数的任何改变都不会影响到调用此函数的程序内对应变量的原始值。

- 传地址

  此方法是将参数的地址复制给函数的形式参数。在函数中，通过传入的参数地址，形式参数可以直接改变实际变量的内容，此实际变量是在调用此函数的程序内使用的。因此改变形式参数可以连带改变变量的内容。

→ **函数的返回值**

函数可以利用 **return** 语句将数值返回至调用此函数的程序。返回值必须是函数所指定的数据类型，如果 *return-type* 是 **void** 类型即表示没有返回值，应该没有数值在 **return** 语句之中。执行到 **return** 语句之后，函数会回到调用此函数的地方继续执行，任何在 **return** 语句之后的语句都不会被执行。

指针与数组

指针

指针是存有另一变量地址的变量，例如，如果一个指针变量 varpoint 存放变量 var 的地址，则 varpoint 指向 var，宣告指针变量的语法如下：

```
data-type *var_name;
```

指针的 *data-type* 需是合法的 C 数据类型，它标明了 *var_name* 所指向的变量的数据类型。在 *var_name* 之前的星号（*）是告知 C 编译器 *var_name* 为一指针变量。有两个特殊运算符（*）和（&）与指针的使用有关，例如在变量之前加上&运算符可以存取此变量的地址，而在变量之前加上*运算符则可取得此变量所指地址的内容。

除了*和&之外，还有四个运算符可以使用于指针变量，分别是+、++、- 和 . 。只有整数值才能加到指针变量或从指针变量减去。另外，当执行指针的加减运算时，指针的值会依据它所指向的数据类型的长度而调整。

数组

数组是具有相同数据类型而且可用同样名称使用的变量列表。在数组内的各变量被称为数组元素，数组的第一个元素是定义在下标为 0 的元素而最后一个元素是定义在下标为元素总数减 1 的元素。C 编译器会将一维数组(one-dimension)安置在地址连续的存储器中，第一个元素放在最小的地址。C 编译器不对数组做边界检查。

不支持将一个数组赋值给另一个数组的运算，必须从第一个数组以一次一个元素的方式复制到第二个数组对应的元素。只要是变量或常量可以使用的地方，就可以使用数组元素。

结构体与共用体(Structures and Unions)

→ **结构体**

- 语法

```
struct struct-name
{
      data-type member1;
      data-type member2;
      ...
      data-type membern;
} [variable-list];
```

- 说明

结构体是一或多个相同或不相同数据类型的变量的集合，整合在单一名称下以方便处理。结构体可以被复制、赋值或传递给函数、也可由函数返回。C 编译器支持位的数据类型及嵌套式结构体。

保留字 **struct** 表示要定义一个结构体，而 *struct-name* 为此结构体的名称，在结构体中 *data-type* 必须是合法的数据类型，在结构体中的成员可以定义为不同的数据类型。*variable-list* 宣告为 *struct-name* 类型的变量，而结构体中的每一个项目是一个成员。在定义一个结构体之后，其它相同类型的变量可以使用下列语法进行宣告：

```
struct struct-name variable-list;
```

要存取结构体的成员时，必须指定变量名称及结构体的成员名称，中间加上句点分隔。语法如下：

```
variable.member1
```

variable 是结构体类型的变量而 *member1* 是结构体成员的名称。一个结构体成员的数据类型可以是前面已经定义好的结构体，这种结构即所谓的嵌套结构(nested structure)。

- 范例

```
struct person_id
{
      char id_num [6];
      char name [3];
      unsigned long birth_date;
} mark;
```

→ **共用体**

- 语法

```
union union-name
{
     data-type member1;
     data-type member2;
     ...
     data-type memberm;
} [variable-list];
```

- 说明

  共用体是将不同类型的变量聚集为一群并使用相同的存储器空间。共用体类型类似于结构体类型，但是对于存储器的使用却极为不同。在结构体中所有的成员顺序地安排存储器空间，而在共用体类型中，所有的成员都从同一地址安置而且共用体类型的大小等于成员中占用最大空间的类型的大小。存取共用体类型成员的方式与存取结构体成员方式相同。

  union 是一个保留字而 *union-name* 为此共用体的名称,*variable-list* 定义有相同数据类型的变量，可有可无。

- 范例

```
union common_area
{
     char name [ 3 ];
     int id;
     long data;
} cdata;
```

前置处理伪指令

前置处理伪指令将会指导编译器如何去编译源程序代码。此伪指令类似一个简单的宏处理器，在编译器正式编译源程序前先行处理某些程序。一般而言，前置处理伪指令不会直接编译成执行码。C 编译器在编译的初期会将源程序中的前置处理指令行移除并做适当的处理，同时也会将调用宏指令的程序替换为宏展开后的程序，以及其它的数据，例如**#line** 命令。前置处理伪指令以#号做为开头，即以#号开头的程序行即被视为前置处理伪指令，其后则为命令的名称，以下为前置处理伪指令：

→ **宏替换：#define**

- 语法

```
#define name replaced-text
#define name [(parameter-list)] replaced-text
```

- 说明

#define 伪指令定义字符串常量。在编译源程序之前，会以定义的字符串常量替换到程序中。其主要目的是增加源程序的可读性与维护性。如果无法在一行中写完 *replaced-text*，可以使用反斜线(\)表示还有更多的程序行。

- 范例

```
#define TOTAL_COUNT   40
#define USERNAME      Henry
#define MAX(a,b)      (((a)>(b))?(a):(b))
#define SWAP(a,b)     {int tmp;\
                      tmp=a;\
                      b=a;\
                      a=tmp;}
```

→ **#error**

- 语法

```
#error “message-string”
```

- 说明

#error 伪指令会生成一个使用者所定义的诊断信息，*message-string*。

- 范例

```
#if         TOTAL_COUNT > 100
#error      “Too many count.”
#endif
```

→ **条件编译：#if　#else　#endif**

- 语法

```
#if  expression
     source codes1
[#else
     source codes2]
#endif
```

- 说明

#if 和**#endif** 是一组用来做条件编译程序的伪指令，而编译条件则取决于 *expression* 的运算值。**#else** 伪指令提供二选一的编译方式，它是可有可无的。如果 *expression* 运算的结果不为零，则 *source codes1* 将被编译，否则如果有 *source codes2*，则 *source code2* 被编译。

- 范例

```
#define MODE 2
#if MODE > 0
     #define DISP_MODE MODE
#else
     #define DISP_MODE 7
#endif
```

→ **条件编译：#ifdef**

- 语法

```
#ifdef symbol
     source codes1
[#ifdef
     source codes2]
#endif
```

- 说明

#ifdef 伪指令类似**#if** 伪指令，但是它不是以表达式的结果决定编译的程序行，而是以检查所指定的 *symbol* 是否已经被定义的方式决定的。**#else** 伪指令提供二选一的编译方式，它不是一定要有的。如果 *symbol* 已经被定义则 *source codes1* 将被编译，否则如果 *source codes2* 存在，它将被编译。

- 范例

```
#ifdef DEBUG_MODE
#define TOTAL_COUNT 100
#endif
```

→ **条件编译：#ifndef**

- 语法

```
#ifndef symbol
     source codes1
[#else
     source codes2]
#endif
```

- 说明

#ifndef 伪指令与**#ifdef** 伪指令类似。**#else** 伪指令提供二选一的编译方式，它是可有可无的。如果 *symbol* 没有被定义则 *source codes1* 将被编译，否则如果 *source codes2* 存在，它将被编译。

- 范例

```
#ifndef DEBUG_MODE
#define TOTAL_COUNT 50
#endif
```

→ **条件编译：#elif**

- 语法

```
#if   expression1
      source codes1
#elif     expression2
      source codes2
[#else
      source codes3]
#endif
```

- 说明

#elif 伪指令是要随**#if** 伪指令一起使用。除了通常使用的两种条件编译外，它提供第三种条件编译方式。如果 *expression1* 为非零，则 *source codes1* 被编译，如果 *expression1* 为零，则 *expression2* 被检查是否为非零，如果是，则 *source codes2* 被编译，否则，如果 *source codes3* 存在，则 *source codes3* 被编译。

- 范例

```
#if MODE==1
#define DISP_MODE 1
#elif MODE==2
#define DISP_MODE 7
#endif
```

→ **条件编译：defined**

- 语法

```
#if defined symbol
      source code1
[#else
      source code2
#endif
```

- 说明

单操作数的运算符 **defined** 可以使用在**#if** 或**#elif** 伪指令中。如下列格式的流程控制行 #ifdef symbol 则等于#if defined symbol 的效果，而#ifndef symbol 则等于#if !defined symbol 的效果。

- 范例

```
#if defined DEBUG_MODE
#define TOTAL_COUNT 50
#endif
```

→ **条件编译：#undef**

- 语法

  #undef *symbol*

- 说明

  #undef 伪指令会将先前已定义的符号清除，相当于此符号没有被定义。符号一旦被定义，就会一直保持在定义的状态中，直到编译程序的结尾或者使用**#undef** 伪指令取消定义。

- 范例

```
#define TOTAL_COUNT 100
...
#undef TOTAL_COUNT
#define TOTAL_COUNT 50
```

→ **文件包含：#include**

- 语法

  #include <*file-name*>
  or
  #include "*file-name*"

- 说明

  #include 会将指定文件的内容插入到源程序文件中。当使用<*file-name*>的格式时，编译器会从环境变量 INCLUDE 所指定的路径中寻找 *file-name* 文件，如果没有定义 INCLUDE，C 编译器会在指定的路径中搜寻文件。如果使用"*file-name*"的格式，则 C 编译器会以指定的方式搜寻 *file-name* 文件，如果没有指定路径，则会从当前所在的路径中找寻文件。

- 范例

```
#include <ht48c10-1.h>
#include "my.h"
```

盛群 C 语言的扩充功能与限制

盛群 C 语言提供 ANSIC 以外的许多扩充功能，大部分是供给盛群公司各类型单片机使用的。但是由于单片机的资源有限，必须遵守某些限制。

关键字

以下为盛群 C 语言中可使用的关键字：

@ **bit** **norambank** **rambank0** **vector**

下面的关键字与限定词是不能使用的：

double **float** **Register**

存储器区块(memory bank)

对于地址在较高的储存区块（非区块 0）的变量而言，必须使用间接寻址模式去存取它，程序编译后的指令数及执行效果比较不佳。为了达成更大的效益，可以将程序中经常使用到的变量定义在数据存储器的储存区块 0。盛群 C 语言提供关键字 **rambank0** 用来宣告变量在储存区块 0。

- 语法

```
#pragma rambank0
//data declarations
#pragma norambank
```

- 说明

  rambank0 会指示盛群 C 编译器将其后的变量定位于储存区块 0 中，直到出现 **norambank** 关键字或是到程序的结束。对于只有单一数据存储器区块的单片机，这两个关键字则无效。

- 范例

```
#pragma rambank0
unsigned int i, j;      // 变量 i, j 放在储存区块 0
long len;               // 变量 len 放在储存区块 0

#pragma norambank
unsigned int iflag      // 变量 iflag 的储存区块不确定
                        // 因为 norambank
#pragma rambank0
int tmp;                // 变量 tmp 放在储存区块 0
...
i=1;                    //MOV A, 1 (编译后的汇编语言指令)
                        //MOV _i, A

iflag=1                 //MOV A, BANK_iflag
                        //MOV [04H], A
                        //MOV A, OFFSET_iflag
                        //MOV [03H], A
                        //MOV A, 1
                        //MOV [02H], A
```

位数据类型

盛群 C 语言提供位数据类型，此类型可用于变量的宣告、参数列表以及函数的返回值。位变量的宣告与其它 C 数据类型的变量宣告一样。对于具有多 RAM/ROM 储存区块的单片机，应该将位变量宣告在 RAM 储存区块 0（**#pragma rambank0**）。

- 范例

```
#pragma rambank0
bit test_flag;            // 位变量应该放在储存区块 0

bit testfunc(             //bit function
bit f1,                   //bit arguments
bit f2)
{
     ...
     return 0;            //return bit value
}
```

- 限制
  - 为了利用位数据类型的优点，不建议将变量宣告为位数组的数据类型。
  - 指针不可设定为位类型。

内嵌式汇编语言

- 语法

```
#asm
[label:] opcode [operands]
...
#endasm
```

- 说明

  #asm 和**#endasm** 是内嵌式汇编语言的前置处理程序伪指令。C 编译器会将 **#asm** 之后的(或夹在**#asm** 和**#endasm** 之间的)汇编语言指令直接写进输出的文件，有如直接使用汇编语言撰写程序。

- 范例

```
//convert low nibble value in the accumulator to ASCII
#asm
; this is an inline assembly comment
and a, 0fh
sub a, 09h
sz c add a, 40h-30h-9
add a, 30h+9
#endasm
```

中断

盛群 C 语言提供一种使用伪指令**#pragma** 以实现中断服务程序的方法（ISRs）。伪指令**#pragma vector** 用来宣告 ISR 的名称与存储器地址，之后若有函数的名称与**#pragma vector** 定义的符号名称相同时，此函数就是这个中断向量的中断服务程序。在中断服务程序中的 return 语句将会编译成 **RETI** 指令。

- 语法

  #pragma vector *symbol @ address*

- 说明

  symbol 是中断服务程序的名称，*address* 是中断地址，复位向量（地址 0）固定由主函数 main( )使用，任何中断服务程序不可使用此中断地址。

- 限制

  撰写 ISR 程序时要注意有四种限制：

  – ISR 没有输入参数且返回类型是 **void**。

  – ISR 不能够重复进入，且而在 ISR 中不可让任何中断再发生。

  – 在程序中不要直接调用 ISR 程序，应该由中断信号输入时自行调用它。

  – 在 ISR 中不要调用用 C 写的函数。但是可以调用系统函数或 C 编译器内建的函数(built-in function) 。如果必须在 ISR 中调用函数，可以使用汇编语言撰写这个函数。

- 范例

```
#pragma vector timer0 @ 0×8
extern void ASM_FUNCTION();
void setbusy(){
...
}

void timer0(){
     ...
     ASM_FUNCTION();    //The ASM_FUNCTION should be an
                        //assembly function
     _delay(3)          //Ok; built-in function
     setbusy();         //Wrong! Do not call function
}
```

变量

运算符“ @ ”用来指定数据存储器中变量的地址。

- 语法

  data_type variable_name @ memory_location

- 说明

  memory_location 指定变量所在的地址。在只有单一 RAM/ROM 存储器区块的单片机中，*memory_location* 是一个字节，而在具有多个 RAM/ROM 存储器区块的单片机中，*memory_location* 是两个字节，高字节存放存储器区块的编号。请参考盛群单片机的资料手册以取得 RAM 存储器空间的信息。

- 范例

```
int v1 @ 0×5B;     // 宣告变量 v1 放置于 RAM bank 0, offset 0×5B
int v2 @ 0×2F0;    // 宣告变量 v2 放置于 RAM bank 2, offset 0×F0
```

静态变量

盛群 C 语言提供有效范围在文件内的静态变量，而不支持局部的静态变量。

- 范例

```
static i;               // 宣告静态变量, 以文件为有效范围
void  f1 () {
      i=1;              // OK 可以使用此变量
}
void f2 () {
      static int j;     // 错误的宣告,不能将函数中的局部变量宣告为静
                           态变量
                        // local static variable is not supported
      ...
}
```

常量

盛群 C 语言支持二进制常量，任何以 0b 或 0B 开头的字符串将被视为二进制常量。

- 范例

```
0b101=5
0b1110=14
```

函数

避免撰写重复进入与递归的程序行。

数组

任何数组应该配置在一个连续的存储器区内，而且最大不可超过 256 个元素。严格来说，数组的大小取决于所使用单片机的数据存储器区的大小。

常量

常量必须宣告为全局型且在宣告时就要设定初始值。常量不可宣告为外部使用。数组常量需要指定数组的大小，否则会产生错误。

```
const char carray [ ]={1,2,3}; // 错误,没有指定数组的大小
const char carray [3]={1,2,3}; // 正确
```

字符串常量必须在包含 main( ) 主函数的 C 语言文件中使用。

```
//test.c
void f1 (char *s);
void f2 () {
      f1 (“abcd”)    //“abcd” 是字符串常量
                     // 如果在文件 test.c 中没有定义 main( )主函数
                     // 则 Holtek C 编译器会发出错误信息
      ...
}
...
void main(){
      ...
}
```

指针

指针不能用于常量与位变量。

初始值

全局变量宣告时不可以同时设定初始值，局部变量则无此项限制，但是常量在宣告时则一定要设定初始值。

- 范例

```
unsigned int i1=0;        // 错误; 全局变量, 不可设定初始值

unsigned int i2;          // 正确
const unsigned int i3;    // 错误; 常量, 必须设定初始值

const unsigned int i4=5; // 正确
const char a1[5];         // 错误; 数组常量, 必须设定初始值

const char a2[5]={0×1 0×2 0×3 0×4 0×5};  // 正确
const char a2[4]=“abc”;  //={‘a’, ‘b’, ‘c’, 0}
const char a2[3]=“abc”;  //={‘a’, ‘b’, ‘c’}
const char a2[2]=“abc”;  // 数组大小不一致
```

乘法/除法/模

乘法、除法和模（“ * ”，“ / ”，“ % ”）运算符由系统调用执行。

内建函数

- WDT & halt & nop

C 系统调用	汇编语言码
void_clrwdt ( )	CLR WDT
void_clrwdt1 ( )	CLR WDT1
void_clrwdt2 ( )	CLR WDT2
void_halt ( )	HALT
void_nop ( )	NOP

- 左移/右移

```
void_rr (int*);      //rotate 8 bits data right
void_rrc (int*);     //rotate 8 bits data right through carry
void_lrr (long*);    //rotate 16 bits data right
void_lrrc (long*);   //rotate 16 bits data right through carry
void_rl (int*);      //rotate 8 bits data left
void_rlc (int*);     //rotate 8 bits data left through carry
void_lrl (long*);    //rotate 16 bits data left
void_lrlc (long*);   //rotate 16 bits data left through carry
```

- 高/低半字节的交换

```
void_swap (int*);    //swap nibbles of 8 bits data
```

- 以指令周期为单位的延迟函数

```
void_delay(unsigned long);   //delay n instruction cycle
```

_delay 函数强迫单片机去执行所指定的周期数。周期数为零则会执行无穷的循环。

_delay 函数的参数只能为常量值并不接受变量。

- 范例 1

```
// 假设 watch dog timer 看门狗定时器已经启动
// 看门狗定时器的清除指令选择为使用一条清除指令
void error (){
     delay (0);           // 无穷的循环，类似 while(1);
}
void dotest(){
     unsigned int ui;
     ui =0×1;
     rr(&ui);             //rotate right
     if (ui != (unsigned int)0×80) error();
     ui =0×ab;
     swap(&ui);
     if (ui != (unsigned int)0×ba) error();
}
```

```
void main(){
          unsigned int i;
          for(i=0; i<100; i++){
                _clrwdt();
                _delay(10);    // 延迟 10 个指令周期
                dotest();
          }
     }
```

- 范例 2

```
     // 假设 watch dog timer 看门狗定时器已经启动
     // 看门狗定时器的清除指令选择为使用两条清除指令
     void do test(){
     ...
     }
     void main(){
          unsigned int i;
          for(i=0; i<100; i++){
                _clrwdt1();
                _clrwdt2();
                dotest();
          }
     }
```

堆栈

因为盛群单片机堆栈的层数是有限的，所以要考虑函数调用时的层数以避免堆栈溢出。乘法、除法、取模和常量是使用 “ call ” 指令实现其功能的，都只占用一层的堆栈。

运算符/系统函数	所需要的堆栈层数
main()	0
_clrwdt()	0
_clrwdt1()	0
_clrwdt2()	0
_halt()	0
_nop()	0
_rr(int*);	0
_rrc(int*);	0
_lrr(long*);	0
_lrrc(long*);	0
_rl(int*);	0
_rlc(int*);	0
_lrl(long*);	0
_lrlc(long*);	0
_delay(unsigned long)	1
*	1
/	1
%	1
constant array	1

第十一章

混合语言

11

盛群程序工具组（汇编编译器、连接器、函数库管理器和盛群 C 编译器）为混合语言的编程提供了一些方法，如使用盛群的汇编语言和 C 语言。也就是说，项目中的源程序文件可以同时使用汇编语言和 C 语言去完成。然而，程序设计师在使用这两种语言设计程序时应该遵守一些规则。为了帮助程序的顺利完成，本章将说明盛群 C 编译器在将 C 语言程序译成汇编程序时所遵循的一些常规惯例以及如何定义子程序的名称等。以下即为相关的议题：

- Little endian
- 函数与参数的命名规则
- 参数的传递
- 返回值
- 寄存器内容的保存
- 在 C 程序中调用汇编语言函数
- 在汇编程序中调用 C 函数
- 使用汇编语言编写 ISR 函数

Little Endian

盛群 C 编译器采用 little-Endian 的数据格式，即一个 WORD 的低字节是此 WORD 的最低的字节(least significant byte)，而高字节则是最高的字节(most significant byte)，在存储器的配置中，低字节占用较低的地址，而高字节占用较高的地址。

范例

```
long var @ 0×40;
var = 0×1234;
```

地址 0×40 存放数据 0×34，地址 0×41 存放数据 0×12。

函数与参数的命名规则

盛群的汇编编译器（Assembler）在处理符号名称时是不分大小写的。事实上，所有的符号名称不管原来的形式为何，都将被译成大写字母。但是盛群的 C 语言则有大小写之分。为了区分这两种语言的不同，凡是定义在 C 语言文件内而且被汇编程序使用到的变量及函数，都必须以大写字母来命名。

当 C 编译器将程序译成为汇编语言时，会在全局变量与 C 语言函数的名称前加上前缀下划线(underscore)。至于局部变量，如果只是宣告但未使用的局部变量，C 编译器不会为其保留存储器空间。可查看 C 编译器所生成的汇编语言文件，以找出 C 局部变量在编译后的名称。

全局变量

在 C 文件中的全局变量，编译后不会改变其名称大小写，但会在前缀加上下划线。

范例

```
TimerCt
TMP
```

编译后

```
_TimerCt
_TMP
```

局部变量

在 C 函数中的局部变量如果没有被其它程序所使用，则不会被译成汇编语言。可查看汇编语言文件找出结果为何。

```
void main(){
      int i, j, k;          ; k 未被使用
      long m;
      char c;
      i = j = m = c = 2;
      #asm
      set CR3[1].2          ; set bit 10 of m, i.e. m|= 0×400
      #endasm
}
```

汇编语言文件中对应的部分如下：

```
#line 2 "C:\Holtek IDE\SAMPLE\NAME.C"
LOCAL CR1 DB ? ; i
#pragma debug variable 2 CR1 i
#line 2 "C:\Holtek IDE\SAMPLE\NAME.C"
LOCAL CR2 DB ? ; j
#pragma debug variable 2 CR2 j
#line 3 "C:\Holtek IDE\SAMPLE\NAME.C"
LOCAL CR3 DB 2 DUP (?) ; m
#pragma debug variable 2 CR3 m
#line 4 "C:\Holtek IDE\SAMPLE\NAME.C"
LOCAL CR4 DB ? ; c
#pragma debug variable 2 CR4 c
```

第二与第三行表示 *i* 被译成汇编语言中的 *CR1*，同理 *j* 被译成为 *CR2*，*m* 被译成为 *CR3* 且 *c* 被译成为 *CR4*，*k* 没有被使用所以没有被编译。

注意: 如果加入新的局部变量或调整局部变量的顺序，那么编译后的名称会有所改变。

针对上述的范例，如果是具有多个数据存储器区块(memory bank)的单片机，则指令

set CR3 [1].2

可能会执行正确也可能错误。如果 CR3 被配置在较高编号的存储器区块，则程序会执行错误，但是这种现象是不可能发生的。因为局部变量被译成汇编语言时，会使用 **LOCAL** 伪指令定义此变量，同时指示编译器将变量配置在数据存储器的区块 0，因此就像定义在单一存储器区块的变量一样，可以正确的执行。

函数

就像全局变量一样，在 C 文件中的函数名称在编译时并不会改变字形的大小字，只是会加上前缀下划线。

范例

```
GetKey
IsBusy
```

译成

```
_GetKey
_IsBusy
```

函数的参数

C 文件中函数参数的名称被译成函数名称加上参数出现的序号，序号是从 0 开始计算的。

范例

```
GetKey (int row, long col)
```

row 被译成为 GetKey0

col 被译成为 GetKey1

参数的传递

由于单片机的资源限制，盛群 C 编译器通过 RAM 存储器取代堆栈传递参数至函数，函数参数的命名是以函数名称附加由 0 开始到所有参数数目的数字。就像局部变量一样，函数参数也是配置在数据存储器的区块 0。

范例

```
void function (int a, int b)
```

参数 a 将被译成为 function0，参数 b 将被译为 function1。使用两种程序语言写程序时，即使函数参数的数据类型超过一个字节，在汇编语言中仍应该把它宣告为 BYTE 的数据类型，例如参数为 WORD（2 个字节）的数据类型时，应该使用“ DB 2 DUP(?) ”来宣告。

返回值

C 函数的返回值会被存于 **A** 寄存器或 **RH** 系统变量。如果返回值的大小是一个字节（例如 **char**、**unsigned char**、**int**、**unsigned int**、**short**、**unsigned short**），则返回值被储存于 **A** 寄存器，如果返回值的大小是二个字节（例如 **long**、**unsigned long**、**pointer**）则高字节被储存于 **RH** 而低字节被储存于 **A** 寄存器。

注意: **RH** 变量位于 RAM 存储器的区块 0。

寄存器内容的保存

除了 ISR 函数，在所有使用汇编语言设计的函数中，不需要保存寄存器的内容。如果要用汇编语言设计 ISR 函数，则使用者有责任将 ISR 中所用到的寄存器的内容保存起来，当 ISR 函数执行完毕回到被中断之处前，将原先的数据回存给这些寄存器。

在 C 程序中调用汇编语言函数

此节将描述在 C 程序中调用汇编语言函数的步骤。此步骤共分为两个部分，一部分为汇编语言程序文件，另一部分为 C 语言程序文件。

→ **在汇编语言文件中：**

- 如果返回值为二个字节，则将 **RH** 宣告为外部字节变量。
- 将以下划线为前缀的函数名称宣告为公用函数(public)。
- 如果函数有参数，将这些参数宣告在数据存储器的区块 0，以及宣告为公用的参数。注意参数的命名。
- 将返回值放入 **A** 寄存器或 **RH** 系统变量。

→ **在 C 语言文件中：**

- 将以大写字母命名的函数宣告为外部函数(external)。
- 调用此函数。

下面的函数是定义在汇编语言文件中，被 C 程序所调用。

```
long KEYIN( int row, long col );
```

汇编语言文件：

```
;; 将变量 RH 宣告为外部变量
EXTERN RH:BYTE

;; 将函数与参数宣告为公用的
PUBLIC _KEYIN, KEYIN0, KEYIN1

;; 将函数参数安排在存储器的区块 0
RAMBANK0 KEYINDATA          ; 假设单片机具有多个 ram 区块(bank)
KEYINDATA .section ‘data’
KEYIN0 DB?                  ;row
KEYIN1 DB 2 DUP (?)         ;col, 不要使用 ‘KEYIN1 DW ?’

;function body
CODE .section ‘code’
_KEYIN:
...
MOV A, KEYIN0               ;取得参数 row 值
...
MOV A, KEYIN1               ;取得参数 col 的低字节
...
MOV A, KEYIN1[ 1 ]          ;取得参数 col 的高字节
...
;;将返回值放入寄存器 A 及系统变量 RH 中
MOV A, 0A0H                 ; 假设返回值是 0×A010
MOV RH, A                   ; 将高字节 0×A0 存入变量 RH
MOV A, 10H                  ; 将低字节 0×10 存入寄存器 A
RET
```

在 C 语言文件中：

```
// 将被调用的函数宣告为外部函数,同时以大写字母对此函数命名
extern long KEYIN(int row, long col);
long rc;
...
// 调用外部函数
rc = KEYIN(10, 20L);
```

在汇编程序中调用 C 函数

此节将描述在汇编程序中调用 C 函数的步骤。针对具有多个 ROM 存储器区块的单片机，在调用函数之前，必须要设定 BP 寄存器（存储器区块指针）。

→ **在 C 语言文件中**

- 宣告以大写字母命名的函数。

→ **在汇编语言文件中：**

- 如果返回值是二个字节，则宣告 **RH** 为外部的字节变量。
- 宣告以下划线为前缀命名的函数为外部函数。
- 如果函数需要参数，则将这些参数宣告为外部参数，要注意参数的命名规则。
- 如果函数有参数，将输入值设定给参数。
- 调用 C 函数。如果单片机只具备一个 RAM/ROM 存储器区块，则直接调用 C 函数。如果单片机具备多个 RAM/ROM 存储器区块，则先将 BP 寄存器设为函数所在的存储器区块，再调用 C 函数。

- 从 **A** 寄存器或系统变量 **RH** 中取出返回值。

下面的函数是以 C 语言撰写而被汇编程序所调用。

long KEYIN(int row, int col);

当单片机只具备单一的 ROM 存储器区块。

```
//-------------------------------------------------
// In C file, 定义函数
//-------------------------------------------------
long KEYIN(int row, long col){
...
}
void main( ){
...
}
;;-------------------------------------------------
;; In assembly file
;;-------------------------------------------------
;; 宣告 RH 为外部变量
EXTERN RH:BYTE

;; 宣告以下划线为前缀的函数名为外部函数
extern _KEYIN:near    ;;underscore and function name

;; 将函数的参数宣告为外部变量
extern _KEYIN0:byte   ; 参数 row
extern _KEYIN1:byte   ; 参数 col, 虽然其数据类型为 2 个 bytes,
                      ; 还是将其宣告为 BYTE

code_ki .section ‘code’

;; 设定参数的输入值,准备调用函数 KEYIN(0×10, 0×200L)
mov a, 10H
mov KEYIN0, a         ; 将参数 row 设定为 10H
mov a, 2H
mov KEYIN1[1], a      ; 设定参数 col 高字节为 02H,低字节为 00H
clr KEYIN1            ;

;; 调用 C 的函数
call _KEYIN

;; 从寄存器 A 或系统变量 RH 中取得返回值
;; 寄存器 A 存放返回值的低字节,系统变量 RH 存放返回值的高字节
```

下面的例子中，函数定义在 C 语言文件中而被汇编程序所调用。

long KEYIN(int row, int col);

此单片机具有多个 ROM 存储器区块。

```
//---------------------------------------------------
// In C file, 定义函数
//---------------------------------------------------
long KEYIN(int row, long col){
...
}

;---------------------------------------------------
; In assembly file
;---------------------------------------------------

; 宣告 RH 为外部变量
    EXTERN RH:BYTE

    ; 宣告以下划线为前缀的函数名为外部函数
    extern_KEYIN:near

;; 将函数的参数宣告为外部变量
    extern_KEYIN0:byte  ; 参数 row
    extern_KEYIN1:byte  ; 参数 col, 虽然它是 2 bytes,
                        ; 但是仍要宣告为 BYTE

    code_ki  .section  'code'

    ;; 设定参数的输入值, 准备调用函数 KEYIN(0×10, 0×200L)
    mov a, 10
    mov KEYIN0,a        ; 参数 row
    mov a, 2
    mov KEYIN1[ 1 ],a   ; 参数 col 的高字节
    clr KEYIN1          ; 参数 col 的低字节

    ;;调用位于多个 ROM 存储器区块的函数
    ;;先要将 BP 寄存器设定为函数所在的存储器区块编号
    mov a, bank _KEYIN
    mov bp, a           ; 更改区块编号 bank number
    call _KEYIN         ; 调用函数

    ;; 从寄存器 A 或系统变量 RH 中取得返回值
    ;; 寄存器 A 存放返回值的低字节,系统变量 RH 存放返回值的高字节
```

使用汇编语言撰写 ISR 函数

ISR（中断服务程序）是由硬件中断所调用，不可由程序自行调用，因此不会有参数的输入，也不会有值返回。如果用汇编语言撰写 ISR 函数，则不会与 C 语言的程序有关系，只需要将此汇编语言文件加入项目即可。若需要更多关于撰写 ISR 程序的资料，请参考汇编语言的使用说明。

不论 ISR 是由汇编语言或 C 语言所撰写，都不要从 ISR 中调用 C 函数。

第十二章

连接器

12

连接器的作用

连接器会将汇编编译器或 C 编译器所生成的目标文件连接起来生成任务文件。它会结合目标文件中的程序代码和数据，并且从函数库文件中搜寻公用函数或变量以解决源程序中所使用的外部函数或外部变量。如果没有直接指定地址，连接器也会在 ROM/RAM 存储器的指定地址或默认的地址为程序段及数据段定位。最后，连接器会将程序代码和其它数据输出到任务文件(task file)，此任务文件由盛群 IDE（集成开发环境）载入到盛群 HT-ICE（硬件仿真器）中执行除错。连接器所引用的函数文件是由盛群函数库管理器生成的。

连接器选项

选项将会控制连接器执行所指定的任务。在第三章 Options 菜单中，Project 命令内提供一个连接器选项对话框，可以设定这些选项。包括：

函数库文件

- 语法
  libfile1[,*libfile2*...]
  这个选项告知连接器，如果输入的目标文件中使用到未定义的程序或变量时，就从指定的函数库文件中寻找。如果函数库的某个程序模块包含被调用的程序或变量，则只有此程序模块会被包含到输出的任务文件，而不是将整个函数库文件包含进来（参考第十三章函数库管理器）。

程序段地址

- 语法
  section_name=*address*[,*section_name*=*address*]...
  此选项指定段在存储器中的地址；*section_name* 是被寻址的段的名称，段名至少要在一个输入的目标文件中被定义，否则会出现警告信息。address 是被指定的段的地址，其格式为十六进制的 xxxx。

生成地址映射文件

此选项的检查框用来指定是否要生成地址映射文件。

地址映射文件

地址映射文件会列出程序中所有程序段及数据段的名称、存储器地址与长度，也会列出连接文件时遇到的信息。连接器会在此文件的结尾列出程序进入点的地址。地址映射文件还会列出所有公用变量的名称与地址。如果连接器找不到外部变量或程序所对应的公用变量或公用程序，则也会在地址映射文件中记录外部变量或程序的名称及它们所在文件的文件名。地址映射文件的内容如下所示：

```
Holtek (R) Cross Linker Version 7.34
Copyright (C) HOLTEK Microelectronics INC. 2002-2003.All
rights reserved.
Input Object File: C:\SAMPLE\T2.OBJ
Input Object File: C:\SAMPLE\T1.OBJ
Start End   Length  Class    Name
0000h 00F2h 00F3h   CODE     TEXT   (C:\SAMPLE\T1.OBJ)
00F3h 0114h 0022h   CODE     SUB    (C:\SAMPLE\T2.OBJ)
0000h 0063h 0064h   DATA     DAT    (C:\SAMPLE\T1.OBJ)

Address Public by Name
001Ch  BREAKL
00A4h  CHKSTACK
0042h  FAC_DB

Address Public by Value
001Ch  BREAKL
0042h  FAC_DB
00A4h  CHKSTACK

HLINK: Program entry point at section code (address 0) of file
       C:\SAMPLE\T1.OBJ
<EOF>
```

连接器的任务文件与除错文件

连接器输出的文件之一是任务文件，它由文件标题和二进制数据两个部分组成。文件标题部分包含有连接器的版本、单片机名称型号和程序存储器的大小，二进制数据部分则是程序代码。连接器输出的另一个文件是除错文件，它记录所有盛群 IDE 除错程序所需要的数据，包括源程序文件的文件名、符号的名称和源文件中程序行的行号。盛群 IDE 系统会参考这些符号除错的数据。除非已完成除错程序，否则不要将此文件删除，避免盛群 IDE 系统无法提供符号除错的功能。

第三部份

公用程序

除了先前讨论的通用 8 位单片机开发工具外，对于特殊用途的语音和 LCD 单片机，通过所提供的仿真工具及对声音合成和语调生成器执行仿真的逐步导引，加上 LCD 面板的及时仿真。盛群提供一些额外的公用程序。这个部份包含能够快速及有效的开发程序和除错所需要的信息。

第十三章

函数库管理器

13

函数库管理器的功能

函数库管理器提供处理函数库文件的功能。连接器(Linker)在生成输出文件的过程中会使用到函数库文件。函数库是由一个或多个被编译过的目标模块所组成，而且是在连接文件时使用。此文件存有其它程序可能需要的执行模块。

可使用函数库管理器去建立函数库文件，并将包含有公用程序的目标文件加到函数库文件中。在生成这些目标文件之前，必须利用汇编语言的伪指令 PUBLIC 将所有可共享的程序设定为公用程序（可参考汇编语言和编译器等章节）。然后使用编译器将源程序文件编译成目标文件（.OBJ），再经函数库管理器将目标文件加到所指定的函数库文件中。在连接的过程中，如果连接器发现在程序中有未解决的名称时（也就是程序中使用到外部变量或函数，但是在所有输入的目标文件之中找不到同名的公用变量或函数），它会从函数库文件中找寻这些未解决的名称，并且将包含这些名称的程序模块取出及复制。假如在函数库模块中找到未解决的名称，连接器会将此模块连接到输出的任务文件内。

设定函数库文件

函数库管理器提供以下的功能：

- 生成新的函数库文件
- 往函数库文件中添加程序模块或从函数库文件中删除程序模块
- 从函数库文件中取得一个程序模块并且生成包含此模块的目标文件

如图 13-1 选用 Tools 菜单的 Library Manamger 命令。图 13-2 显示函数库管理器的功能对话框。

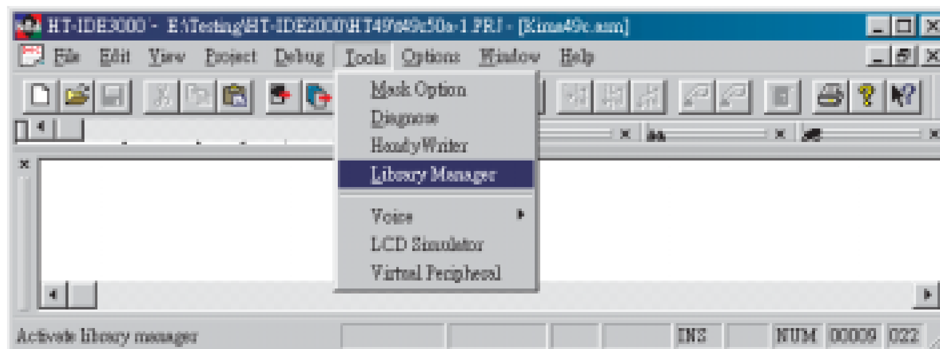


图 13-1

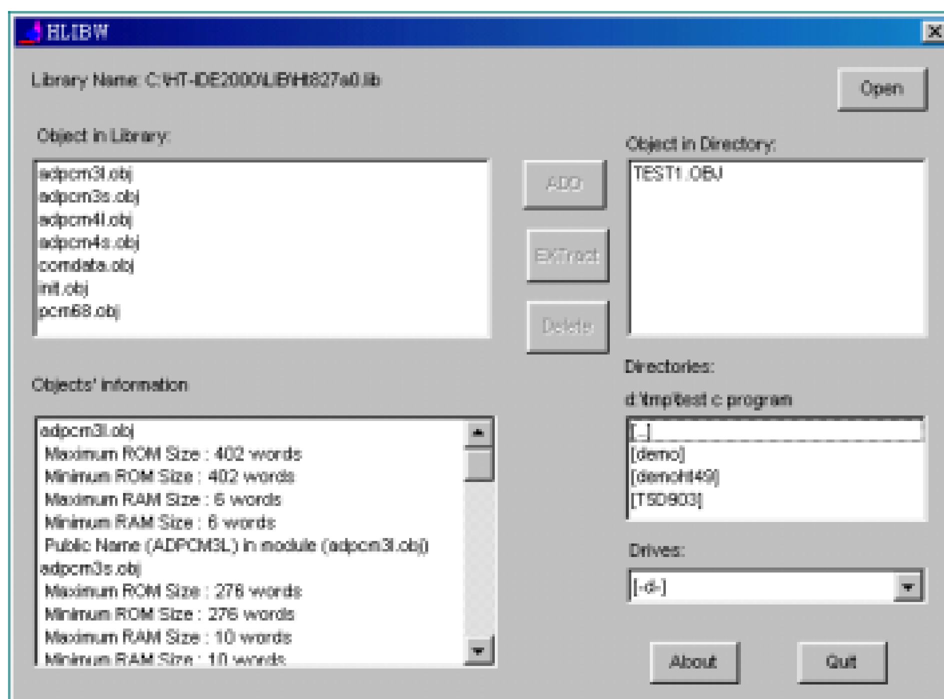


图 13-2

生成新的函数库文件

按下图 13-2 右上角的 Open 按钮，会出现图 13-3。

在 File Name [N]处键入一个新的函数库文件的文件名，并按下 OK 按钮，会出现图 13-4 的对话框以便确认。假如选择 YES 按钮，将会生成一个新的函数库文件，但是不包含任何程序模块。

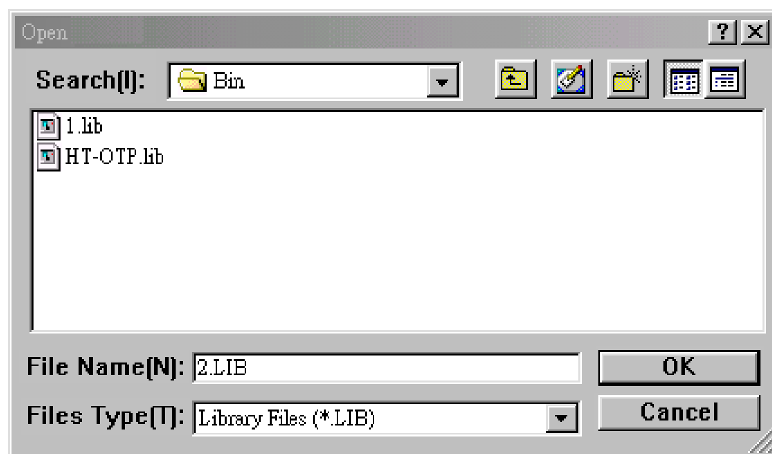


图 13-3



图 13-4

往函数库文件中添加程序模块

从“ Object in Directory ”对话框中选择一个目标模块，并按下[ADD]钮将这个目标模块加到函数库文件中。

从函数库文件中删除程序模块

从“ Object in Library ”框中选择一个目标模块，并按下[Delete]钮从函数库文件中将目标模块删除。

从函数库文件中取得程序模块并生成目标文件

从“ Object in Library ”框中选择一个目标模块，并按下[ExTract]钮。之后会生成一个与目标模块同名而且有相同内容的文件，这个文件的文件名会显示在“ Object in Directory ”框中。

目标模块的信息

按下 Open 钮，将出现图 13-3 所示的画面。在 File Name 对话框上方的列表框中选择一个函数库文件，按下 OK 钮。在图 13-2 中，所选择的函数库文件中所有的目标模块会被列在“ Object in Library ”框中。与每个目标模块有关的信息也会列在“ Objects′ Information ”框中，详述如下：

- 最大的程序存储器容量(ROM)
  目标模块程序代码使用的最大存储器容量，与程序段(code section)的位置类型有关。
- 最小的程序存储器容量(ROM)
  目标模块程序代码实际使用的最小容量。
- 最大的数据存储器容量(RAM)
  目标模块程序数据使用的最大容量，与数据段(data section)的位置类型有关。
- 最小的数据存储器容量(RAM)
  目标模块程序数据实际使用的最小容量。
- 公用名称
  目标模块中所有公用符号的名称。

第十四章

LCD 仿真器

14

简介

盛群的 LCD 仿真器，即是 HT-LCDS，提供 LCD 驱动器的软件仿真。根据使用者设计的图形和控制程序，HT-LCDS 在显示器上实时显示这些图形，在尚未取得真实的 LCD 硬件面板之前，LCD 仿真器将使开发的过程更为便利。但是如果项目所用的单片机没有支持 LCD 功能，则这些命令是无效的。

LCD 面板配置文件

在开始 LCD 的仿真之前，必须先建立一个 LCD 面板配置文件。HT-LCDS 将从 LCD 面板配置文件取得 LCD 的设定数据并根据仿真时取得的数据将图形显示在显示器上，如果这个文件不存在，HT-LCDS 将无法仿真 LCD 的动作。对于具有处理 LCD 功能的单片机,必须要建立 LCD 面板配置文件以便执行 LCD 的仿真。工具菜单(Tools menu)中的 LCD Simulator 命令可用来建立面板配置文件及执行 LCD 的仿真（图 14-1）。LCD 面板配置文件包含面板的配置数据及图形信息。

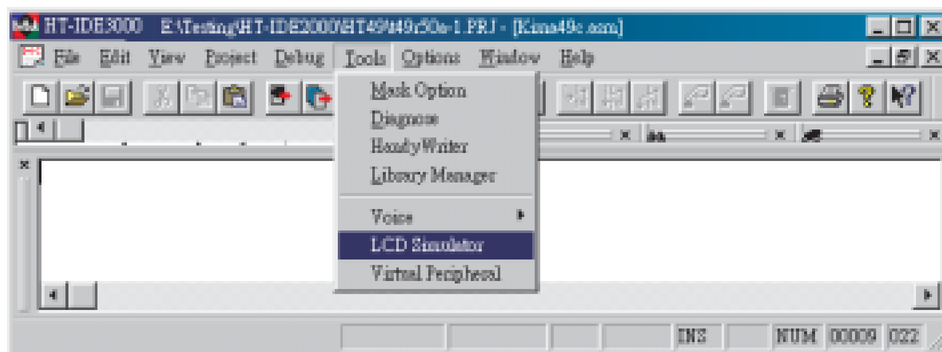


图 14-1

面板文件的文件名与项目的关系

原则上，除了后缀名为.lcd 外，面板配置文件的主文件名和当前项目(current project)的名称是相同的。HT-LCDS 会认为这个文件是对应于当前项目的面板配置文件。使用 HT-LCDS 的 File 菜单中的 New 命令或工具栏的 New 按钮可以建立新的面板配置文件。如果想要将面板配置文件改为与当前项目不同的文件名时，可以使用 File 菜单的 Save 命令或工具栏中的 Save 按钮。

当 HT-LCDS 开始仿真时，它要参考当前的面板配置文件以获取仿真所需要的信息，此时可由选择 HT-LCDS 的 File 菜单的 New 或 Open 命令将面板配置文件启动为当前文件。LCD 面板配置文件的文件名可能和当前项目的名称相同，也可以选用不同的名称。

选择 HT-LCDS

从 Tools 菜单中选用 LCD simulator 命令时，如果当前项目对应的面板配置文件已存在，如图 14-2 的窗口会出现。在此窗口下方的表格中，某些指定的 COM/SEG 位置中会显示所对应的位图图形文件(bitmap)的文件名，同时这些图形会显示在窗口上方的面板显示框。如果在项目的路径中并未存有相关的面板配置文件，则 LCD 仿真器在启动时并不会显示面板显示框和 COM/SEG 表格。图 14-3 是 HT-LCDS 的工具栏信息。

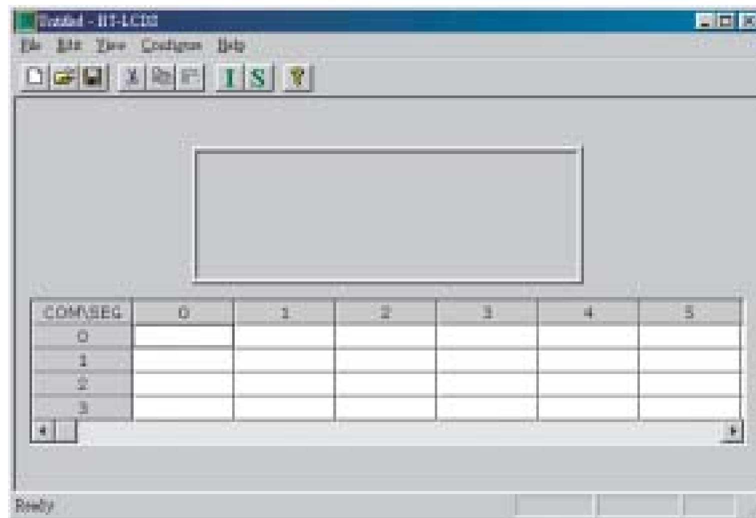


图 14-2

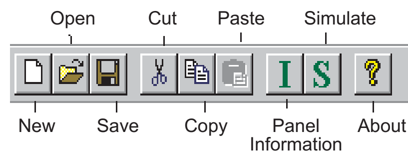


图 14-3

New： 建立一个新的面板配置文件。
Open： 打开一个已存在的面板配置文件。
Save： 储存面板配置文件。
Cut： 删除一个图形。
Copy： 复制一个图形到剪贴板。
Paste： 加入已复制的图形到控制板上。
I： 面板信息对话框。
S： 进入 LCD 仿真模式。

LCD 面板图形文件

LCD 面板中的图形是一种位图文件（.bmp），用来表示面板上实际的图形和它们在面板上的位置，任何位图编辑器都可以用来建立此种文件。HT-LCDS 中 Edit 菜单的 Panel Editor 命令也可以用来设定 LCD 面板图形的信息。位图文件可有可无，当 LCD 面板是点矩阵形式时，是不需要图形文件的。

建立 LCD 面板配置文件

以下两个步骤用来建立一个面板配置文件。

- 设定面板的配置，包括 LCD 驱动器的 segment 和 common 的个数、面板的宽度和高度，也可以设定面板配置文件的路径以及点矩阵形式的面板。
- 选择图形和它们的位置，这将建立图形与 COM/SEG 位置间的关系。

建立面板的配置结构

使用 HT-LCDS 中 File 菜单的 New 命令建立面板的配置信息。面板配置对话框会显示（图 14-4）。设定正确的 LCD 规格数据，如 COM/SEG 的个数、面板的宽度、高度和图形文件所在的路径，之后按下[OK]钮。完成面板配置的设定后，系统会回到图 14-2 以便选取图形。

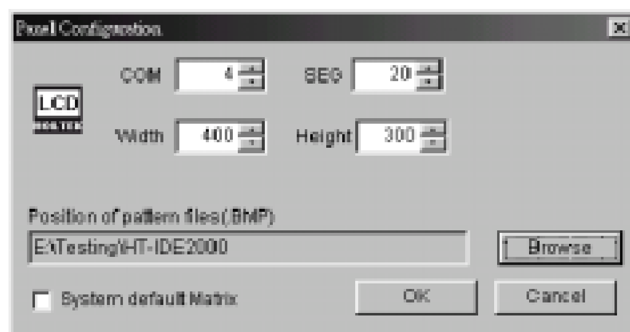


图 14-4

面板的配置包括：

- COM 和 SEG。设定 LCD 驱动器的 COMMON 个数和 SEGMENT 个数。当图 14-4 出现时，会显示出单片机中 LCD 驱动器的默认个数。必须确认这些个数和单片机中 LCD 驱动器的实际个数是一致的。
- 宽度和高度。这是面板显示在显示器上的尺寸。可以改变此项以调整面板画面的大小。
- 面板配置文件的路径。使用 Browse 钮选择面板配置文件所存放的路径或设定在项目的路径。
- 点矩阵形式。仿真点矩阵形式的 LCD 面板。图 14-5 显示点矩阵屏幕。

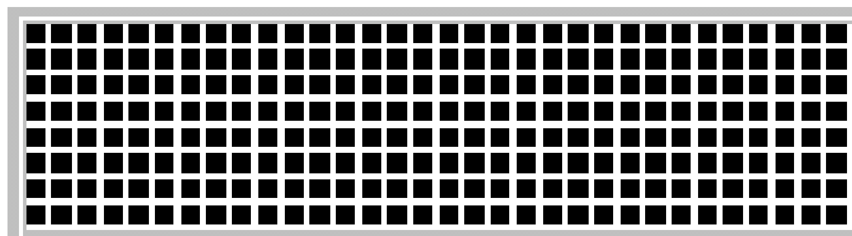


图 14-5

注意: 不要设定与实际 LCD 驱动器不相同的 COM 和 SEG 个数，否则会发生不可预测的结果。

选择图形并设定位置

底下的步骤说明如何选择图形和位置。

- 使用 HT-LCDS 中 File 菜单的 New 命令建立一个新的面板配置文件，完成面板配置的设定后，图 14-2 会显示。接着需要从 Pattern Information 对话框（图 14-6）中选择图形以及设定 COM/SEG 位置。在加入新的图形的一节中会有更详细的介绍。
- 使用 HT-LCDS 中 File 菜单 Open 命令打开一个存在的面板配置文件。所有的图形都会显示在图 14-2 上方的面板显示框内，而图形文件的文件名会出现在图 14-2 中 COM/SEG 的表格中。使用者可以加入，删除或改变图形信息、包括图形的文件名和位置。
- 使用 HT-LCDS 中 Edit 菜单的 Panel Editor 命令打开一个面板图形文件，如果这个面板图形文件已经被设定，便不需要选择图形，只需设定图形的位置。使用 Panel Editor 定义图形的一节中将对此部分有详细的描述。

加入新的图形

- 移动光标到如图 14-2 中 COM/SEG 位置的格子中,并点击鼠标 2 次，将会出现如图 14-6 所示的 Pattern Information 对话框。Pattern List 列表框会将此项目路径中所有的图形文件（.bmp）排列显示。Size 方格中是所选图形的位图大小，Com 和 Seg 方格中是这个图形被设定的 COM/SEG 位置，这三个方格中的数据是无法修改的。
- 从 Pattern List 列表框中选择一个图形文件（一个位图文件）或使用 Browse 钮从其它路径中选取一个图形文件。HT-LCDS 使用双色的位图文件当作图形的影像来源。Preview 窗口会将所选取的图形放大显示。
- 设定图形在面板显示框内的 X/Y 坐标位置。
- 按下[OK]钮回到图 14-2，然后点击 File 菜单的 Save 命令或在点击工具栏上的 Save 钮，这样，面板文件的建立或修改就完成了。

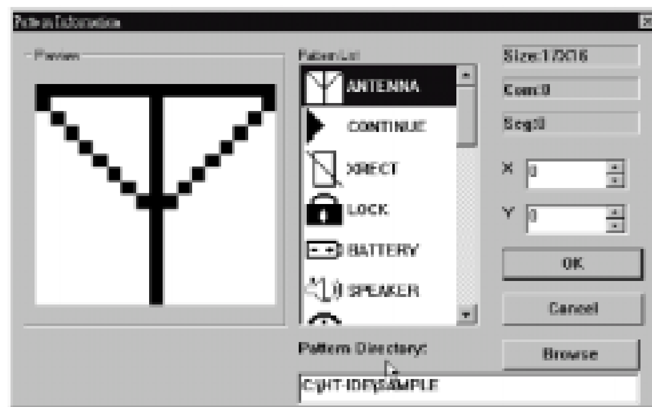


图 14-6

删除图形

- 如图 14-2 所示，选取欲删除图形的 COM/SEG 并按下[Delete]钮或点击工具栏上的 Cut 钮。

改变图形

- 首先删除要改变的图形，然后加入一个新的图形以更换之。
- 另外一种方式，在图 14-2 中，选定图形的 COM/SEG 位置并双击鼠标，图 14-6 Pattern Information 对话框会出现，从这个对话框中的 Pattern List 列表框内选取一个图形并且按下[OK]钮。

改变图形位置

- 如图 14-2 所示，可使用点击并拖拉(Select-drag-drop)的方法直接将图形移到面板显示框的适当位置。
- 另外一种方法，在图 14-2 中，选定图形的 COM/SEG 位置并双击鼠标，图 14-6 Pattern Information 对话框会出现，在这个对话框中的 X 及 Y 方格内设定图形的新位置并且按下[OK]钮。

当以上的操作都已完成且系统回到如图 14-2 的窗口时，点击 HT-LCDS 中 File 菜单的 Save 命令或点击工具栏上的 Save 钮，完成面板文件的建立或修改。

如何加入用户定义的矩阵

在 COM/SEG 个数与面板的 ROW/COL 个数不一致时，HT-LCDS 提供一个映射策略（File 菜单的 Import user matrix 命令）以帮助定义一个新的矩阵，例如：假设一个 2 COMs 和 6 SEGs 的 LCD 面板要使用 3 ROWs × 4 COLs 的矩阵，规划成下列的对应表：

COM0-SEG0	COM0-SEG1	COM0-SEG2	COM0-SEG3
COM1-SEG0	COM1-SEG1	COM1-SEG2	COM1-SEG3
COM0-SEG4	COM0-SEG5	COM1-SEG4	COM1-SEG5

上面的矩阵可用底下的定义文件定义：

```
; MATRIX.DEF
; Comment line
ROW=3
COLUMN=4
; mapping syntax：ROW,COL=>COM,SEG
0,0 => 0,0 ; Map Row0 co10 to COM0 SEG0
0,1 => 0,1 ; Map Row0 co11 to COM0 SEG1
0,2 => 0,2 ; Map Row0 co12 to COM0 SEG2
0,3 => 0,3 ; Map Row0 co13 to COM0 SEG3
1,0 => 1,0 ; Map Row0 co10 to COM1 SEG0
1,1 => 1,1 ; Map Row0 co11 to COM1 SEG1
1,2 => 1,2 ; Map Row0 co12 to COM1 SEG2
1,3 => 1,3 ; Map Row0 co13 to COM1 SEG3
2,0 => 0,4 ; Map Row0 co10 to COM0 SEG4
2,1 => 0,5 ; Map Row0 co11 to COM0 SEG5
2,2 => 1,4 ; Map Row0 co12 to COM1 SEG4
2,3 => 1,5 ; Map Row0 co13 to COM1 SEG5
```

使用 Panel Editor 定义图形

HT-LCDS 提供一个完整的面板编辑接口来定义 LCD 面板的图形，如果已经存在整个面板的图形文件，就可以不用在面板上去设定所有图形的文件名，只需要去设定这些图形在面板上的位置。

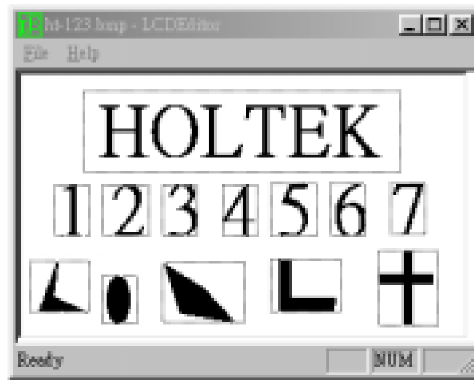


图 14-7

底下的步骤说明如何设定图形在 LCD 面板上的位置

- 在完成面板配置的设定之后，使用 Edit 菜单的 Panel Editor 命令进入 Panel Editor 窗口。
- 在 Panel Editor 窗口中使用 File 菜单的 Open 命令打开面板图形文件(.bmp)。

注意: **Panel Editor** 只支持双色的.BMP 文件(黑白)

- 面板图形文件中所有的图形会显示在如图 14-7 的 Panel Editor 窗口中。
- 双击鼠标或使用按下拖放的方法，在 COM/SEG 字段中设定显示的图形文件名。在 Save Pattern 对话框出现后就可以输入图形的数据。
- 针对面板上的每个图形重复执行上述的步骤。
- 在完成所有图形的数据设定后，回到 Panel Editor 窗口。再使用 File 菜单的 Save 命令将这些设定储存。
- 离开 Panel Editor 窗口回到 HT-LCDS 窗口，此时面板上会显示新的设定。

使用批处理文件将图形加入面板

通过 Edit 菜单的 Add Item Batch 命令，HT-LCDS 可从批处理文件中将图形加入面板。批处理文件是后缀名为.BTH 的文字文件，在批处理文件中每个图形项目需要定义图形文件的文件名和图形在面板中的位置。当使用 Menu 菜单的 Add Item Batch 命令选取一个批处理文件后，HT-LCDS 会将此批处理文件中描述的所有图形加入到面板中的指定位置。底下是一个.BTH 文件的例子。

```
; this is a comment line.
; item syntax: BMPfile.bmp, COM, SEG, X, Y
CRYSTAL.BMP,      0, 2, 120, 30
FION.BMP,         2, 3, 200, 50
CLIN.BMP,         3, 2, 130, 90
STEVE.BMP,        4, 4, 20,  40
```

选择 LCD 面板的颜色

HT-LCDS 提供如图 14-8 所示的调色对话框，可使用 HT-LCDS 中 Configure 菜单的 Set Panel Color 命令来选择面板的颜色。

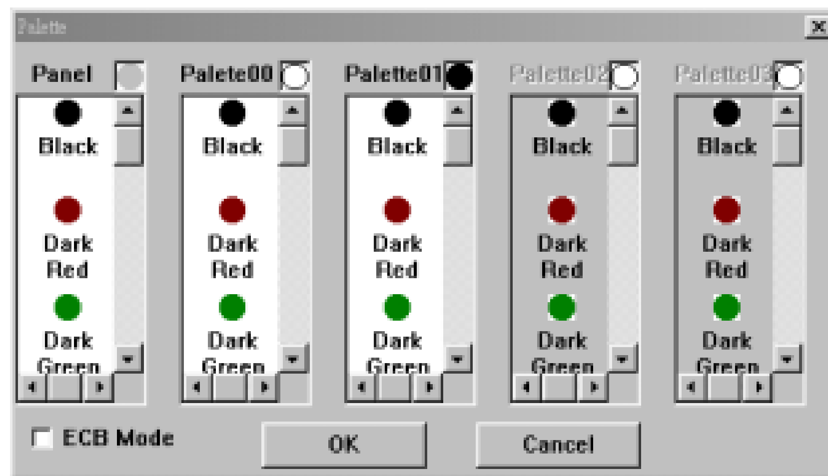


图 14-8

注意： ECB 模式只针对 HTG21×0 的彩色 LCD

LCD 仿真

在开始 LCD 仿真之前，确定 HT-LCDS 参照到正确的面板配置文件。然后如图 14-1 和 14-2 所示，使用 Tools 菜单的 LCD Simulator 命令即可进入 HT-LCDS 环境。

- 点击工具栏的 S 钮，HT-LCDS 会参阅对应的面板配置文件开始进行 LCD 的仿真。
- 如果打开了一个并非当前项目对应的面板配置文件，并点击工具栏的 S 钮，则 HT-LCDS 会参阅这个打开的面板配置文件并进行 LCD 的仿真。

当 HT-LCDS 开始仿真时将会显示如图 14-9 的窗口，此时最新状态的 LCD 图形会出现在面板显示框之中。

停止仿真

用鼠标双击 LCD 仿真窗口的名称栏将使 HT-LCDS 回到编辑模式。

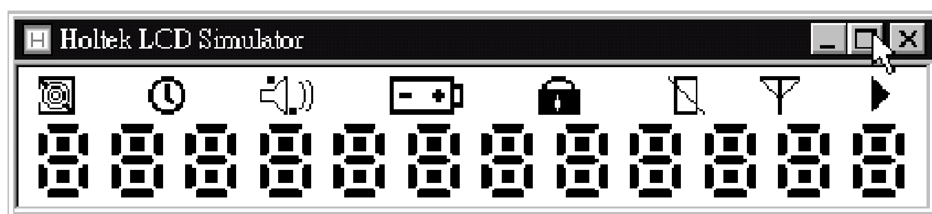


图 14-9

第十五章

虚拟外围设备管理器 (VPM) 15

简介

在大部分的应用中，单片机会连接到一些外部的硬件设备以实现所需要的功能，然而在软件仿真的过程中要牵涉到一些外部的硬件，则超出了单片机软件仿真器的仿真能力范围。为了克服这个问题，盛群开发出虚拟外围设备管理器（VPM），它可以让使用者将外部的外围设备加入到单片机的项目中去。结合 HT-IDE 软件仿真器，VPM 让使用者直接驱动和监控这些外部硬件设备的输入和输出，进而更有效的除错和实作应用产品。

VPM 窗口

图 15-1 显示一个 VPM 窗口的实例。如同大部份的窗口，VPM 窗口包含执行功能菜单的工具栏与显示程序信息的状态栏，并在主窗口画面显示已经加到项目中的外围设备。

已经加入项目的外围设备是被认知的 VPM 的组件，可用鼠标左键在这些需要的组件上点击选取。在文件中，被选取的组件将被视为现有组件(current component)。用鼠标双击现有组件，将显示连接对话框，在此对话框中可以设定外围设备与单片机之间的连接关系。若使用鼠标右键点击某些现有组件，则会显示结构对话框，可用来设定组件的特性。

状态栏中包含模式（Mode）、现有组件（Current Component）、时间（Time）和周期（Cycle）等四个部分。模式部分指出 VPM 当前处于结构模式(configuration)或是执行模式(running)。现有组件部分指出现有组件的名称。时间部分指出 VPM 执行模式时的总执行时间，周期部分指出 VPM 在执行模式时的总周期计数。

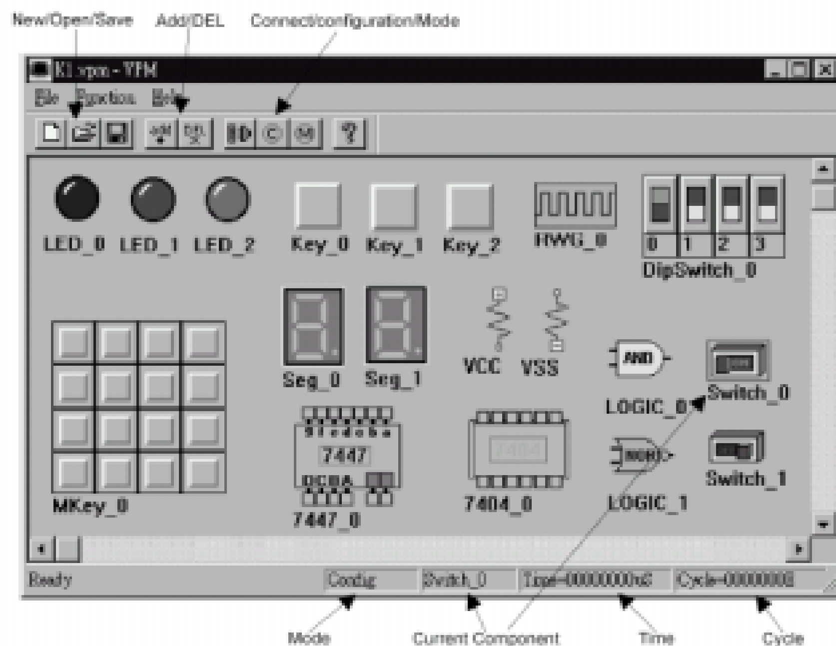


图 15-1

VPM 菜单

文件菜单

File 菜单上有六个功能（如图 15-2），在工具栏上有三个对应的主要功能（如图 15-3）。

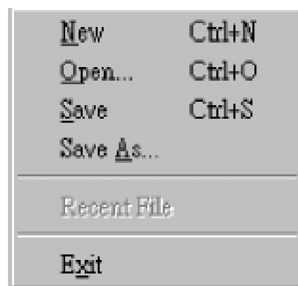


图 15-2



图 15-3

→ **New**

建立新的 VPM 项目文件。每次 VPM 启动时，系统会自动建立新的 VPM 项目文件。

→ **Open**

打开已存在的 VPM 项目文件。

→ **Save**

储存当前的项目文件。

→ **Save As**

以不同的名称储存当前的项目文件。

→ **Exit**

退出 VPM。

功能菜单

在 Function 菜单上有五个项目（如图 15-4）。在工具栏上也可找到相对应的按钮（如图 15-5）。



图 15-4

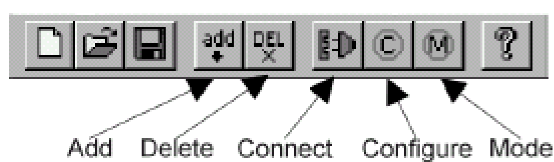


图 15-5

→ **Add**

加入一个新的外围设备到项目中。按下工具栏的 Add 钮，则会出现 Add Peripheral 对话框（如图 15-6）。选择所要的外围设备再按 OK 钮。

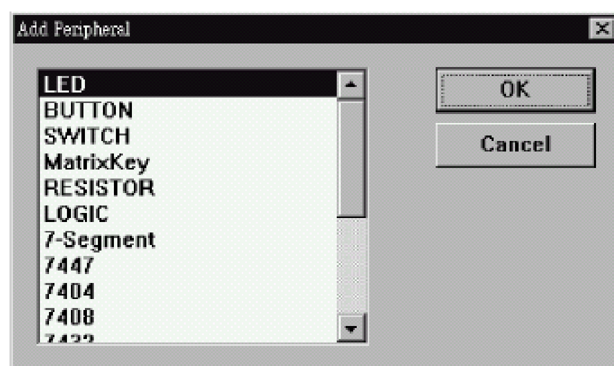


图 15-6

→ **Delete**

从项目中删除外围设备。先选择所要删除的组件，然后按工具栏上的 Del 钮，则被选择的组件会被删除。

→　**Connect**

选择所要连接的组件，然后按工具栏上的 Connect 钮，会出现连接对话框，如图 15-7。对话框中的连接状态路径框显示此组件的连接状态，使者可用 Connect/Disconnect 钮来进行连接或断开连接。

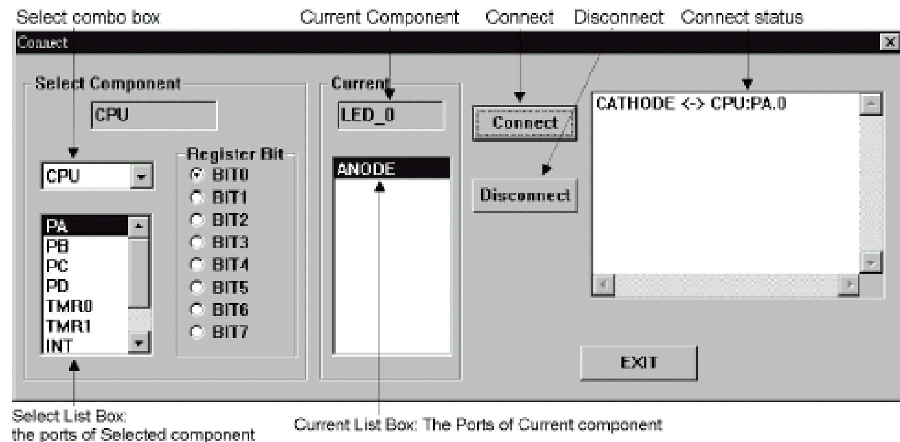


图 15-7

以图 15-7 为例，它显示名为 LED_0 的 LED 组件的连接对话框。以此例来说，现有组件为 LED_0。Select combo 框会将项目中所有可以与 LED_0 连接的组件显示出来。而 Select List 列表框会将所选组件的引脚显示出。Register Bit 则显示详细的引脚。LED 组件有阴极与阳极两根引脚。在此例中 LED_0 的阴极与 CPU 的 A 端口的 bit0 连接。

→ **CONFIG**

某些组件可以设定其特性。先选择所要的组件，再按工具栏的 Configure 钮，若此组件有特性选项可设定，则会出现 CONFIG 对话框。图 15-8 显示了 LED 的 CONFIG 对话框。



图 15-8

→ **Mode**

VPM 具有执行模式与结构模式两种模式。按下工具栏上的 Mode 钮或选择 Function 菜单的 Mode 命令会使 VPM 在这两种模式中切换。在结构模式中，可用 Add/Del/CONFIG 来编辑虚拟的外部电路。在执行模式下，VPM 除了显示盛群 IDE 单片机仿真的结果外，也会根据它们的结构设定显示这些组件的运行结果。

VPM 外围组件

LED



图 15-9

LED 有两个引脚，分别为阴极与阳极。当阴极为 0 阳极为 1 时，LED 被点亮。可在结构对话框(Configure)中设定 LED 的颜色。

Button/Switch

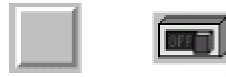


图 15-10

BUTTON/SWITCH 在开路位置有 debounce time 和 switch status 两个选项。Debounce 的单位为毫秒。BUTTON 是具有 non-latching 的瞬间动作，而 SWITCH 则是具有 latching 的非瞬间动作。DipSwitch 可在单一包装中提供多个 switch 的功能，其个数可以调整。

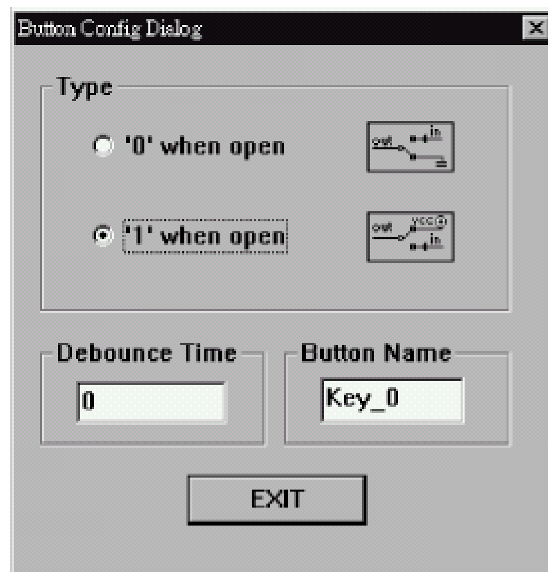


图 15-11

Seven Segment Display

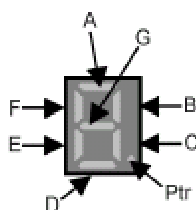


图 15-12

七段显示器由八个 LED 灯组成，分别为 A，B，C，D，E，F，G 与 ptr。每个 LED 的一个引脚会连到单片机同名的输入端口的不同引脚，LED 的另一引脚则接到共同引脚。此共同引脚可以是阳极或阴极，它决定显示的极性。

→ **Resistor**



图 15-13

此电阻组件主要是用来连接 VCC 或 VSS 以便提供电压上拉(pull-up)或电压下拉(pull-down)的功能。可使用相关结构对话框来设定。

→ **Logic gate**

逻辑门提供六种逻辑功能。



图 15-14

使用 Add 命令选择逻辑门。假如显示的逻辑门并非所需要的，可以按下鼠标的右键将当前可支持的逻辑门显示出，如图 15-14 所示。此时可选取适合的逻辑门。Pin Number 用来设定逻辑门的输入脚数。此处所设定的引脚数会反应在连接对话框中的可用引脚数。

→ **Matrix key**

Matrix key 提供一个标准的矩阵按钮组件设备，可以从结构对话框中设定它的大小。以毫秒为单位来设定矩阵中各按钮的 debounce time。可以在矩阵的特性对话框中设定矩阵各列(column)是与 VCC 连接或与 VSS 连接。

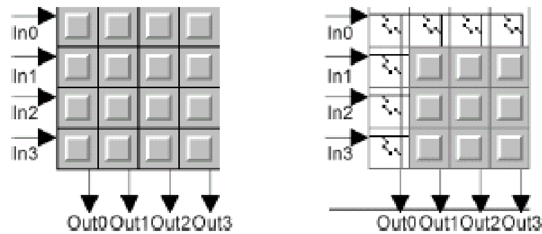


图 15-15

例如当设定行为 4，列为 4 的矩阵按钮，则表示 Matrix key 有 4 个输入脚（行为 4），有 4 个输出脚（列为 4）。

→ **Rectangle wave generator**



图 15-16

方波发生器用来产生方波，其频率和单片机的频率有关。在这个组件的特性对话框中，输入周期表明输入波形的转态所需要的指令周期数，例如若输入周期设为 2，则每 2 个机器周期(machine cycle)方波发生器的输入就会转态一次，这个输入的周期就是所设定的周期值的 2 倍。当选取方波发生器，且双击鼠标左键将显示连接对话框，则此方波发生器只能连接到一个设备。而如果选取被连接的设备且连接对话框显示不止一个设备时，便可连多个设备到同一方波发生器。如果单片机有一个以上的引脚要连接到相同的方波发生器，则需要加入更多的方形发生器去完成。

快速开始的范例

从盛群 IDE3000 使用手册的范例中，挑选一个例子来示范如何构造虚拟外部电路。

霹雳灯

→ **在 HT-IDE 3000 系统中**

- 建立一个新项目，选用 HT48C10-1 单片机（使用 Project/New）
- 将源文件 scanning.asm 加到项目中（使用 Project/Edit）。此文件可在 Holtek IDE\SAMPLE\CHAP15 的路径中找到。
- 将 Holtek IDE 设成软件仿真模式。（使用 Options/Debug/Mode）
- 编译完成此项目。（使用 Project/Build）

→ **在 VPM 中**

- 建立新的 VPM 项目。
- 通过重复按工具栏的 Add 钮以及选取 LED 8 次将 8 个 LED 加到项目中。
- 选择 REGISTER 组件并按下 Add 钮将电阻加到项目中，在刚加入的电阻上双击鼠标左键，再设定电阻的名称为 VCC。
- 将所有的 LED 的阳极接到 VCC，各 LED 的阴极接到单片机 PA 相对的 bit n（n=0~7）。下面示范如何将 LED_0 的阳极连接到 VCC 与 LEC_0 的阴极连接单片机 PA 的 bit 0。
  – 在 LED_0 按下鼠标左键，将 LED_0 选成当前组件。
  – 在 LED_0 按下鼠标右键，会出现连接对话框（如图 15-18）。
  – 将 LED_0 的阴极连接到单片机 PA 的 bit0。
  – 重复以上的步骤，将所有的 LED_n 连到单片机 PA 的 bit_n。
- 按下工具栏上的 Mode 钮将 VPM 由结构模式切换到执行模式。

→ **在 HT-IDE3000 上**

开始除错动作－LED 的输出结果将会显示在 VPM 窗口中。

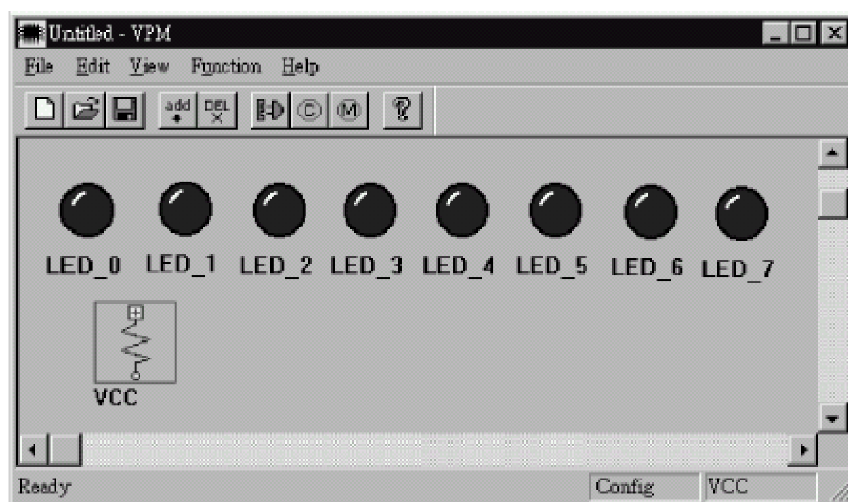


图 15-17

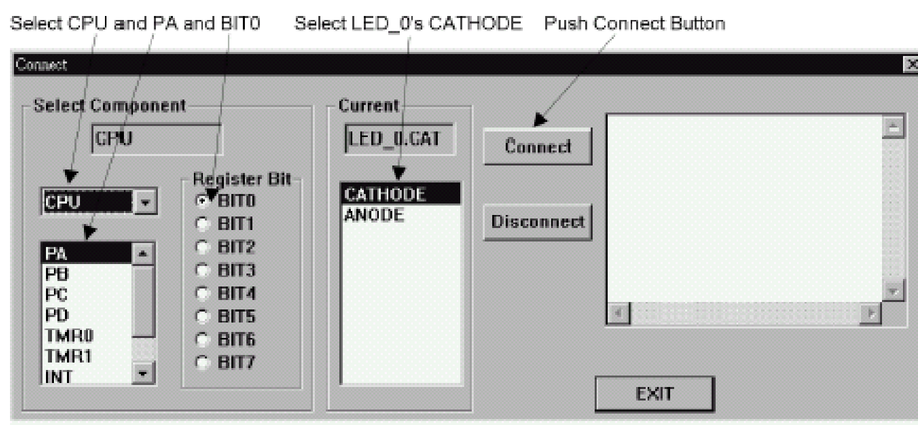


图 15-18

第四部份

附录

附录 A

编译器使用的保留字



汇编语言保留字

下表所列的是使用在汇编语言上的保留字。

- 保留字（伪指令、运算子）

$	DUP	INCLUDE	NOT
*	DW	LABEL	OFFSET
+	ELSE	.LIST	OR
-	END	.LISTINCLUDE	ORG
.	ENDIF	.LISTMACRO	PAGE
/	ENDM	LOCAL	PARA
=	ENDP	LOW	PROC
?	EQU	MACRO	PUBLIC
[ ]	ERRMESSAGE	MESSAGE	RAMBANK
AND	EXTERN	MID	ROMBANK
BANK	HIGH	MOD	.SECTION
BYTE	IF	NEAR	SHL
DB	IFDEF	.NOLIST	SHR
DBIT	IFE	.NOLISTINCLUDE	WORD
DC	IFNDEF	.NOLISTMACRO	XOR

- 保留字（指令助记符）

ADC	HALT	RLCA	SUB
ADCM	INC	RR	SUBM
ADD	INCA	RRA	SWAP
ADDM	JMP	RRC	SWAPA
AND	MOV	RRCA	SZ
ANDM	NOP	SBC	SZA
CALL	OR	SBCM	TABRDC
CLR	ORM	SDZ	TABRDL
CPL	RET	SDZA	XOR
CPLA	RETI	SET	XORM
DAA	RL	SIZ	
DEC	RLA	SIZA	
DECA	RLC	SNZ	

- 保留字（寄存器名称）

A	WDT	WDT1	WDT2

指令集

算术运算指令

ADD A, [m]	ACC 与数据存储器相加，结果放入 ACC
ADDM A, [m]	ACC 与数据存储器相加，结果放入数据存储器
ADD A, x	ACC 与立即数相加，结果放入 ACC
ADC A, [m]	ACC 与数据存储器、进位标志相加，结果放入 ACC
ADCM A, [m]	ACC 与数据存储器、进位标志相加，结果放入数据存储器
SUB A, x	ACC 与立即数相减，结果放入 ACC
SUB A, [m]	ACC 与数据存储器相减，结果放入 ACC
SUBM A, [m]	ACC 与数据存储器相减，结果放入数据存储器
SBC A, [m]	ACC 与数据存储器、进位标志相减，结果放入 ACC
SBCM A, [m]	ACC 与数据存储器、进位标志相减，结果放入数据存储器
DAA [m]	将加法运算中放入 ACC 的值调整为十进制数，并将结果放入数据存储器

逻辑运算指令

AND A, [m]	ACC 与数据存储器做“与”运算，结果放入 ACC
OR A, [m]	ACC 与数据存储器做“或”运算，结果放入 ACC
XOR A, [m]	ACC 与数据存储器做“异或”运算，结果放入 ACC
ANDM A, [m]	ACC 与数据存储器做“与”运算，结果放入数据存储器
ORM A, [m]	ACC 与数据存储器做“或”运算，结果放入数据存储器
XORM A, [m]	ACC 与数据存储器做“异或”运算，结果放入数据存储器
AND A, x	ACC 与立即数做“与”运算，结果放入 ACC
OR A, x	ACC 与立即数做“或”运算，结果放入 ACC
XOR A, x	ACC 与立即数做“异或”运算，结果放入 ACC
CPL [m]	对数据存储器取反，结果放入数据存储器
CPLA [m]	对数据存储器取反，结果放入 ACC

递增和递减指令

INCA [m]	将数据存储器的内容加 1 并将结果放入 ACC
INC [m]	将数据存储器的内容加 1，结果放入数据存储器
DECA [m]	将数据存储器的内容减 1 并将结果放入 ACC
DEC [m]	将数据存储器的内容减 1，结果放入数据存储器

移位指令

RRA [m]	将数据存储器的内容向右移 1 位，结果放入 ACC
RR [m]	将数据存储器的内容向右移 1 位，结果放入数据存储器
RRCA [m]	将数据存储器中的内容和进位标志位一起向右移 1 位，结果放入 ACC
RRC [m]	将数据存储器中的内容和进位标志位一起向右移 1 位，结果放入数据存储器
RLA [m]	将数据存储器的内容向左移 1 位，结果放入 ACC
RL [m]	将数据存储器的内容向左移 1 位，结果放入数据存储器
RLCA [m]	将数据存储器中的内容和进位标志位一起向左移 1 位，结果放入 ACC
RLC [m]	将数据存储器中的内容和进位标志位一起向左移 1 位，结果放入数据存储器

数据传送指令

MOV A, [m]	将数据存储器的内容复制到 ACC 中
MOV [m], A	将 ACC 的内容复制到数据存储器中
MOV A, x	将立即数载入到 ACC 中

位运算指令

CLR [m].i	将数据存储器中的位清除
SET [m].i	将数据存储器中的位置 1

分支指令

JMP addr	无条件跳跃
SZ [m]	如果数据存储器的内容为 0，则跳过下一条指令
SZA [m]	将数据存储器的内容复制到累加器，如果值为 0 则跳过下一条指令
SZ [m].i	如果数据存储器的 i 位为 0，则跳过下一条指令
SNZ [m].i	如果数据存储器的 i 位不为 0，则跳过下一条指令
SIZ [m]	将数据存储器的内容加 1 后，如果结果为 0，则跳过下一条指令
SDZ [m]	将数据存储器的内容减 1 后，如果结果为 0，则跳过下一条指令
SIZA [m]	将数据存储器的内容加 1 后，如果结果为 0，则跳过下一条指令，并将结果存到 ACC
SDZA [m]	将数据存储器的内容减 1 后，如果结果为 0，则跳过下一条指令，并将结果存到 ACC
CALL addr	调用指定地址的子程序
RET	从子程序返回到主程序
RET A, x	从子程序返回到主程序，并载入立即数到累加器
RETI	从中断子程序返回到主程序

查表指令

TABRDC [m]	读取 ROM 码（当前页）到数据存储器和 TBLH 中
TABRDL [m]	读取 ROM 码（最后一页）到数据存储器和 TBLH 中

其它指令

NOP	空操作
CLR [m]	清除数据存储器
SET [m]	置位数据存储器
CLR WDT	清除看门狗定时器
CLR WDT1	预清除看门狗定时器
CLR WDT2	预清除看门狗定时器
SWAP [m]	将数据存储器的低 4 位与高 4 位互换
SWAPA [m]	将数据存储器的低 4 位与高 4 位互换，并将结果存到 ACC
HALT	进入暂停模式

附录 B

编译器的错误信息

B

A0005 **Undefined symbol**
该符号在这个文件中没有被定义。

A0010 **Unexpected symbol**
这个符号是多余的。

A0011 **Symbol already defined elsewhere**
重复定义的符号。编译器不允许重复定义符号。

A0012 **Undefined symbol in EQU directive**
编译器不允许 EQU 伪指令的右边有未定义的符号（即使是前置引用）。

A0013 **Expression syntax error**
表达式的语法错误

A0014 **Cross Assembler internal stack overflow**
编译器处理表达式分析时发生错误。

A0016 **Duplicated MACRO argument**
在 MACRO 定义中，有两个形式参数有相同的名称。

A0017 **Syntax error in MACRO parameters**
MACRO 形式参数（表达式）语法错误。

A0018 **Wrong number of parameters**
MACRO 形式参数的总数与 MACRO 实际参数的总数不相等（参考数目不等于定义数目）。

A0019 **Redefined EQU**
EQU 伪指令的左边符号已经被定义过。

A0020 **Multiple section defined**
这个段的名称和先前定义的段相同。在一个源程序中段的名称必须是唯一的

A0021 **DBIT could be used in data section only**
伪指令 DBIT 只能使用在数据段。

A0022 **DB could be used in data section only**
伪指令 DB 只能使用在数据段。

A0024 **Syntax error**
语句的语法错误。

A0025 **MACRO too deep**
MACRO 嵌套层次过多。嵌套最多为 7 层（递归地调用其它 MACROs）。

A0026 **INCLUDE too deep**
INCLUDE 文件嵌套层次过多。文件嵌套最多为 7 层（递归地包含其它包含文件）。

A0027 **IF too deep**
IF/ENDIF 对的嵌套层次过多。嵌套最多为 7 层。

A0028 **ELSE without IF**
伪指令 ELSE 之前漏掉伪指令 IF（IF/ELSE/ENDIF 不成对）。

A0029 **ELSE after ELSE**
伪指令 ELSE 之后没有伪指令 ENDIF 或 IF（IF/ELSE/ENDIF 有成对）。

A0030 **ENDIF without IF**
伪指令 ENDIF 之前漏掉伪指令 IF（IF/ELSE/ENDIF 不成对）。

A0031 **Open conditional**
条件伪指令（IF/IFE/ENDIF）不成对。

A0032 **（expected**
找不到左边的括号，应该加入到表达式中。

A0033 **ORG overlay**
伪指令 ORG 的存储器地址与先前定义的码重叠。

A0034 **Value out of range**
指定的值超过范围。

A0035 **RAM-space limit exceeded**
数据段的总存储器大小超过所允许的 RAM 大小。

A0036 **ROM-space limit exceeded**
程序段的总存储器大小超过所允许的 ROM 大小。

A0037 **DC could be used in code section only**
伪指令 DC 在数据段不能使用。

A0038 **End of file encountered in MACRO definition**
在 MACRO 的定义中找不到伪指令 ENDM（不对称）。

A0039 **Constant expected**
表达式中需要一个常量。

A0040 **Open procedure**
需要一个伪指令 ENDP 和先前的 PROC 做匹配。

A0041 **Block nesting error**
伪指令 PROC/ENDP 的嵌套是错误的。

A0042 **′ expected**
找不到单引号 ′。

A0043 **Non-digit in number**
标号数包含了非数字符号。

A0044 **EXTERN needs an identifier**
在伪指令 EXTERN 中没有指定标识符。

A0045 **Data type expected**
标识符中的数据类型没有被宣告。

A0046 **Unknown data type**
未知的数据类型。

A0047 **“：” expected**
找不到“：”。

A0048 **Too many local labels**
定义过多的局部标号，两个标号间最多只允许 30 个局部标号。

A0049 **Redefined Section in ROMBANK is inconsistent**
这个段在其它的伪指令 ROMBANK 中已经被宣告。

A0050 **Bank out of range**
伪指令 ROMBANK 中定义的存储器区块数值已超过最大的数。

A0051 **Section Undefined in ROMBANK directive**
伪指令 ROMBANK 中包含一个未定义的段名称。

A0052 **Too many errors**
在编译源文件时发现太多错误。

A0053 **LABEL could be used in data section only**
伪指令 LABEL 不能使用在程序段中。

A0054 **ROMBANK/RAMBANK shall be defined before SECTION is declared**
应该先以伪指令 ROMBANK/RAMBANK 宣告此存储器区块，之后才能定义此存储器区块中的程序段或数据段。

A0055 **Record length overflow**
输出目标文件的记录长度超出范围。

A4001 **Incorrect command line option**
命令行选项错误。

A4002 **Redefined symbol**
指定的符号已经被定义。

A4003 **No source file name**
在命令行中没有源程序文件的名称。

A4004 **Incorrect command line syntax**
命令行语法错误。

A4005 **Could not find file**
找不到指定的文件。

A4007 **Bad instruction format file**
指令说明文件错误。

A4008 **Cross Assembler internal fatal error**
编译器故障，请联络厂商。

A4009 **Out of memory**
没有足够的内存让编译器处理源程序文件。

附录 C

连接器的错误信息



L1001 **No object files specified**
在命令行或批处理文件中没有指定目标文件。
检查命令行的语法是否正确。

L1002 **Object file filename.obj is not found on pass1**
在连接器 pass1 过程中没有找到指定的目标文件 filename.obj。
确认文件（filename.obj）是否在工作路径下，或者联络厂商。

L1003 **Out of memory**
连接器无法取得足够的内存空间。
检查系统可用的内存空间。

L1004 **Illegal section address “dddd”**
命令行的选项/ADDR 所指定的程序段的地址错误。
地址 dddd 应该为十六进制。

L1005 **Illegal command option “option”**
命令行指定的选项（option）错误。

L1006 **Batch file “lbatch.bat” is not found**
找不到指定的批处理文件“lbatch.bat”。
请确认批处理文件（lbatch.bat）是否在工作路径下。

L1007 **Illegal file name “filename.obj”**
指定的文件 filename.obj 包含错误的字符。
更正文件 filename.obj 的字符。

L1008 **Command line syntax error**
命令行的语法错误。

L1009 **Illegal object file “filename.obj”, RecType=xx**
指定的目标文件（filename.obj）格式不正确。
确认这个目标文件是否是通过盛群编译器生成的。

L1010 **Cannot close object file “filename.obj”**
连接器无法关闭指定的目标文件（系统错误）。
联络厂商。

L1011 **Record “xx” H check sum error**
连接器在指定的目标文件中的记录（xxH）中发现校验码误差。
请确认这个目标文件是否由编译器生成。

L1012 **MCU information mismatch**
file “filename1.obj” and “filename2.obj”
两个目标文件在编译时使用了不同的单片机配置。
确认在编译时要使用相同的单片机配置。

L1013 **Library file “libname.lib” does not exist**
指定的函数库文件 libname.lib 不存在，或盛群函数库管理器没有生成该函数库文件。检查函数库文件（libname.lib）是否存在于工作路径下。

L1014 **Cannot close the library file “filename.lib”**
连接器无法关闭指定的文件。
联络厂商。

L1015 **Library file “libname.lib”is not found**
连接器在处理连接工作时无法重新打开所指定的函数库文件 libname.lib。
联络厂商。

L1016 **Object file “filename.obj” is not found on pass2**
在连接器 pass2 过程中找不到指定的目标文件 filename.obj。
联络厂商。

L1017 **Cannot write the checksum of record “xx” H**
连接器无法将记录（xxH）的校验总和码写到输出的文件内。
联络厂商。

L1018 **Cannot write data of record “xx” H**
连接器无法将记录（xxH）数据写到输出的文件内。
检查 PC 的文件系统和工作路径，或联络厂商。

L1019 **Cannot open the debug file “debugname.dbg”**
连接器无法打开除错文件 debugname.dbg。
检查 PC 的文件系统和工作路径，或联络厂商。

L1020 **Cannot open the task file “taskname.tsk”**
连接器无法打开任务文件 taskname.tsk。
检查 PC 的文件系统和工作路径，或联络厂商。

L1021 **Cannot open the map file “mapname.map”**
连接器无法打开图文件 mapname.map。
检查 PC 的文件系统和工作路径，或联络厂商。

L1022 **Cannot create the debug file “debugname.dbg”**
连接器无法生成除错文件 debugname.dbg。
检查 PC 的文件系统和工作路径，或联络厂商。

L1023 **Cannot create the task file “taskname.tsk”**
连接器无法建立任务文件 taskname.tsk。
检查 PC 的文件系统和工作路径，或联络厂商。

L1024 **Cannot create the map file “mapname.map”**
连接器无法建立图文件 mapname.map。
检查 PC 的文件系统和工作路径，或联络厂商。

L1025 **Program code is too large**
程序代码的总和大于单片机 ROM 的大小。
修正程序代码（在 CODE 区域）。

L1026 **Program data is too large**
数据段的总和大于单片机 RAM 的大小。
修正 DATA 段，删除 RAM 中的一些数据。

L1027 **Syntax error in batch file “batch.bat”**
批处理文件中的指令语法不正确。

L1028 **Cannot close the batch file “batch.bat”**
连接器无法关闭指定的批处理文件。
联络厂商。

L1029 **Cannot open the binary file**

L1030 **Cannot create the binary file**

L1031 **Public symbols are duplicated**
Public symbol “sym1” in module “mod-name1”
Public symbol “sym1”in module “mod-name2”
连接器发现名为“sym1”的符号在“mod-name1”和“mod-name2”两模块中均被宣告为公用符号。
将其中一个公用符号改名，所有外部参考此符号的地方也改名。

L1032 **Internal error for File Record**

连接器无法将局部的文件指针转换成全局的文件指针。
联络厂商。

L1033 **Internal error when obtaining the global index**

连接器无法取得全局的文件指针。
联络厂商。

L1034 **Illegal class type for section “sec-name” in the file “filename.obj”**

连接器发现文件（filename.obj）中段（sec-name）的类别名称是错误的（既不是 CODE 也不是 DATA）。
检查此目标文件(filename.obj)是否是 Holtek 的编译器所生成的。

L1035 **Internal error when section “sec-name” of the file “filename.obj” is located**

在分配存储器给段时，连接器无法找到段的名称（sec-name）。
联络厂商。

L1036 **The absolute address for the section is illegal**

段的绝对地址不正确

L1037 **Two sections are overlapping**

Section “sec-name1” in the file “filename1.obj”
Section “sec-name2” in the file “filename2.obj”
在文件“filename1.obj”中分配给段“sec-name1”的 ROM 或 RAM 存储器空间与在文件“filename2.obj”中分配给段“sec-name2”的 ROM 或 RAM 存储器空间重叠。
检查这两个段的地址和长度。
参考编译器生成的列表文件。

L1038 **ROM/RAM (Bank xx) memory allocation failed for section “sec-name” (size) xxH in the file “filename.obj”**

在分配存储器给文件(filename.obj)的公用程序段(sec-name)时，连接器无法寻找到足够的 ROM 或 RAM 空间。
检查输入目标文件中所有程序段的长度，也可检查或修改某些程序段的存储器放置类别以精简程序段使用空间。或者联络厂商。

L1039 **Internal error, failed to get SECDEF**

连接器内部错误。
联络厂商。

L1040 **Illegal ROM bank number**

L1041 **A section in ROM bank is not defined**

L1042 **Failed to move the write pointer for task file**
连接器的内部错误。
联络厂商。

L1043 **Illegal Fixupp record in the file “binary.obj”**
连接器的内部错误。
联络厂商。

L1044 **Illegal LIBHED record in the file**

L1045 **Illegal LIBNAM record in the file**

L1046 **Illegal LIBDIC record in the file**

L1047 **Caller is not a local section**

L1048 **Procedure (“proc-name”)is redefined**

L1049 **Illegal extern index**

L1050 **Local section name not in LNAMES**

L1051 **No corresponding section for extern index**

L1052 **Fail to get the global block ID**

L1053 **MCU RAM information mismatch**

L1054 **Illegal RAM bank number**

L1055 **A section in RAM bank is not defined**

L1056 **Both banks “bank-no1” and “bank-no2” contains the section “sec-name”**

L1057 **Total length of combined sections exceeds the bank size**

L1058 **The specified section address conflicts with the absolute section**

L1059 **The bank number of specified section address conflicts with the section**

L1060 **Error Fixmth data is referred by bank Fixupp**

L2001 **Unresolved external symbol “ext-symbol” in file “filename.obj”**
无法从输入的目标文件或是从指定的函数库文件中找到文件 filename.obj 中名为“ext-symbol”的外部符号。
将定义有 ext-symbol 公用符号的目标文件加到连接器的命令行中（增加输入的目标文件），或将定义有名为 ext-symbol 的公用符号的链接函数库文件含入连接器的命令中。

L2002 **Symbol type mismatch**

Public symbol “symbol1” in module “mod-name1”

External symbol “symbol1” in module “mod-name2”

连接器发现一个外部的符号和一个公用符号具有相同的名称，但有不同的符号类别。

检查此外部符号的符号类别，修改源文件，重新编译并重新连接。

L3001 **Specified section “sec-name” does not exist**

在命令行选项 /ADDR 中所指定的程序段（sec-name）并不存在。

在命令行中输入正确的程序段名称或不要指定此程序段。这是一个警告信息而已，连接器仍会执行存储器的分配工作，犹如没有指定这个选项。

L3002 **Specified address “xxxx” for the code section “sec-name” is illegal**

在命令行选项 /ADDR 中所指定程序段（sec-name）的指定地址有错误（不是一个十六进制的数字或超过存储器的范围）。

在命令行中输入正确的地址或不要指定此程序段。这是一个警告信息，连接器会仍会执行存储器的分配工作，犹如没有指定这个选项。

L3003 **Specified address “xxxx” for the data section “sec-name” is illegal**

L3004 **Recursive situation occurred in procedure “proc-name**”

附录 D

函数库的错误信息

D

U0001 **No library file name**

U0002 **Library file does not exist**

U0003 **Library file exists already**

U0004 **The contents of the library file will be discarded if operation is executed**

U0005 **Can’t open the library file**

U0006 **Can’t create a library file**

U0007 **Can’t create a TMP library file**

U0008 **Incorrect library file**

U0009 **Can’t open the list file**

U0010 **Can’t insert a new module to library**

U0011 **Can’t open the object file**

U0012 **Delete operation fails**

U0013 **Replace operation fails**

U0014 **A module with the same name exists in library already**
在任何的函数库文件中，不可有两个相同名称的模块。
当处理 ADD 操作时函数库管理器将检查这个情况。

U0015 **The module doesn’t exist in library**
指定的模块并不在指定的函数库文件中。
当处理 DELETE , REPLACE , EXTRACT 命令时函数库管理器会检查此情况。

U0016 **No enough memory**
使用者的系统没有足够的内存给函数库管理器使用。

U0017 **Bad object file**
加入到函数库文件的文件，其文件格式不正确。

U0018 **No public name in the specified module**
如果一个符号需要当做公用符号（参考第九章），则需使用伪指令 PUBLIC 宣告为公用符号并重新编译源文件，然后利用函数库管理器将旧的模块取代为新的目标文件。

U0019 **Illegal operation**

U0020 **Fail to close a file**

U0021 **Check sum is incorrect**
函数库管理器的内部错误。

U0022 **Fail to out record to the library file**
函数库管理器的内部错误。

U0023 **Out checksum error**
函数库管理器的内部错误。

U0024 **Fail to seek file**
函数库管理器的内部错误。

附录 E

盛群 C 编译器的错误信息

E

错误码

C1000	Unterminated conditional in **#include**
C1001	Unterminated **#if/#ifdef/#ifndef**
C1002	Unidentifiable control line
C1003	Could not find include file *filename*
C1004	Illegal operator * or & in **#if/#elsif**
C1005	Bad operator (*operator*) in **#if/#elsif**
C1007	**#elif** with no **#if**
C1008	**#elif** after #**else**
C1009	**#else** with no **#if**
C1010	**#else** after **#else** 之后
C1011	**#endif** with no **#if**
C1012	**#define** token is not a name
C1013	**#define** token *token* cannot be redefined
C1014	Incorrect syntax for **"defined"**
C1015	Bad syntax for control line
C1016	Preprocessor control *control* not yet implemented
C1017	Duplicate macro argument
C1018	Syntax error in macro parameters
C1019	macro redefinition on *macro-name*
C1020	Disagreement in number of macro arguments
C1021	EOF in macro argument list
C1022	# not followed by macro parameter
C1024	macro argument is too long
C1025	Unknown internal macro
C1026	Unterminated string or char const
C1027	Undefined expression value

C1028	Bad ?: in **#if/#endif**
C1030	Bad number *number* in **#if/#elsif**
C1031	Empty character constant
C1034	String in **#if/#elsif**
C1035	Syntax error in **#undef**
C1036	Syntax error in **#else**
C1037	Syntax error in **#line**
C1038	Syntax error in **#ifdef/#ifndef**
C1040	Syntax error in **#if/#elsif**
C1042	Syntax error in **#include**
C1043	Syntax error in **#if#endif**
C1044	Syntax error in **#endif**
C1045	Lexical error in preprocessor
C1046	Internal error in **#if/#elsif**
C1047	EOF inside comment
C1048	**#error** directive: “err-string”
C1049	**#line** specifies number out of range
C2001	unrecognized declaration
C2002	invalid use of **auto/register**
C2004	invalid use of *specifier*
C2005	invalid type specification
C2006	invalid use of **typedef**
C2007	missing identifier
C2008	redeclaration of *identifier*
C2009	empty declaration
C2010	invalid storage class
C2011	redeclaration of *identifier* previously declared at file_line_no
C2012	redefinition of *identifier* previously defined at file_line_no
C2013	illegal initialization for *identifier*
C2014	undefined size for type *identifier*
C2015	extraneous identifier *identifier*
C2016	*size* is an illegal array size
C2017	illegal formal parameter types
C2018	missing parameter type
C2019	expecting an identifier
C2020	extraneous old-style parameter list
C2021	illegal initialization for parameter *identifier*
C2022	invalid *operator* field declarations
C2023	missing *operator* tag
C2024	*type* is an illegal bid-field type

C2025 *size* is an illegal bit-field size

C2026 field name missing

C2027 *type* is an illegal field type

C2028 undefined size for field type *identifier*

C2029 size of *type* exceeds *number* bytes

C2030 illegal use of incomplete type *type*

C2031 conflicting argument declarations for function *identifier*

C2032 missing name for parameter *number* in function *identifier*

C2033 undefined size for parameter *type identifier*

C2034 declared parameter *identifier* is missing

C2035 undefined static *type identifier*

C2036 undefined label *identifier*

C2037 expecting an enumerator identifier

C2038 underflow/overflow in value for enumeration constant *identifier*

C2039 unknown enumeration *identifier*

C2040 type error in argument *number* to *identifier*, found *type1* expected *type2*

C2041 too many arguments in *identifier*

C2042 insufficient number of arguments in *identifier*

C2043 unknown size for type *type*

C2044 assignment to const identifier *identifier*

C2045 assignment to const location

C2046 addressable object required

C2047 operands of *identifier* have illegal types *type1* and *type2*

C2048 operand of unary *operator* has illegal type *type*

C2049 syntax error; found *token1* expecting *token2*

C2050 too many errors

C2051 skipping *token*

C2053 invalid operand of unary **&**; *identifier* is declared resister

C2054 invalid type argument *type* to **sizeof**

C2055 **sizeof** applied to a bit field

C2056 cast from *type1* to *type 2* is illegal

C2057 found *type* expected a function

C2059 field name expected

C2060 left operand of –> has incompatible type *type*

C2061 illegal use of type name *type*

C2062 illegal use of argument

C2063 illegal expression

C2064 lvalue required

C2065 unknown field *identifier* of type

C2067 intializer must be constant

C2068 cast from *type1* to *type2* is illegal in constant expressions
C2069 invalid initialization type; found *type1*expected *type2*
C2070 cannot initialize undefined *type*
C2071 missing { in initialization of *type*
C2072 number of initializers not matched
C2073 illegal character @
C2074 invalid hexadecimal constant *identifier*
C2075 invalid binary constant *identifier*
C2076 invalid octal constant *identifier*
C2077 missing *character*
C2078 *idertifier* literal too long
C2079 missing ’
C2080 illegal character *character*
C2081 *idertifir1* is a preprocessing number but an invalid *identifie2 constant*
C2082 invalid floating constant *identifier*
C2083 ill-formed hexadecimal escape sequence
C2084 integer expression must be constant
C2085 illegal **break** statement
C2086 illegal **continue** statement
C2087 illegal **case** statement
C2088 case **label** must be a constant integer expression
C2089 illegal **default** label
C2090 extra **default** label
C2091 extraneous return value
C2092 missing label in **goto**
C2093 unrecognized statement
C2094 illegal statement termination
C2095 redefinition of label *identifier* previously defined at *life_line_no*
C2096 illegal *type* type in **switch** expression
C2097 duplicate **case** label *value*
C2098 illegal return type; found *type1* expected *type2*
C2099 type error: pointer expected
C2100 illegal type “array of *type*”
C2101 missing array size
C2102 type error: array expected
C2103 illegal type *type*
C2104 type error: function expected
C2105 duplicate field name *identifier* in type
C2106 illegal initialization of **extern** *identifier*
C2107 **#endasm** expected

C2109 variable with initialized value must be declared as constant
C2110 ROM constant variable must be initialized
C2111 constant variable must be declared as global
C2112 overflow in octal escape sequence
C2113 bit variable cannot be declared as constant
C2114 unclosed comment
C2115 illegal operation for bit variable
C2116 bit pointer not allowed
C2117 invalid pragma string
C2118 bit member in structure not allowed
C2119 ROM constant variable must not be declared as **extern**
C2120 vector function must not have parameters
C2121 vector function must be **void** type
C2122 const string must be used in the C file having main function
C2123 array should specify the size
C2124 size of "array of *type*" exceeds *n* bypte
C2125 should specify ROM address
C2126 RAM address "@" cannot be used with constant variables
C2127 variable with specific RAM address "@" should be declared as global
C2128 left operand of . has incompatible type
C2129 undeclared identifier
C2130 array size exceeds 255
C2131 more than 255 bytes in *type*
C2132 invalid initial value
C2133 bit array not allowed
C2134 redefinition of vector
C2135 invalid vector
C2136 vector is used
C2137 syntax error; redundant token after **#asm/#endasm**
C2138 in-line asm should be put with a function
C2200 internal error
C2201 insufficient memory
C2202 read error

警告码

C4001	empty declaration
C4002	empty input file
C4003	missing prototype
C4004	inconsistent linkage for *identifier* previously declared at *file_line_no*
C4006	declaration of *identifier* does not match previous declaration at *file_line_no*
C4008	register declaration ignored for *type identifier*
C4009	extraneous 0-width bit field *type identifier* ignored
C4010	more than 127 fields in *type*
C4011	more than 31 parameter in function *identifier*
C4012	old-style function definition for *identifier*
C4013	compatibility of *type1* and *type2* is compiler dependent
C4014	*identifier* is a non-ANSI definition
C4015	missing return value
C4016	static *type identifier* is not referenced
C4017	parameter *type identifier* is not referenced
C4018	local *type identifier* is not referenced
C4019	register declaration ignored for *type identifier*
C4020	more than 127 enumeration constants in *type*
C4021	non-ANSI trailing coma in enumerator list
C4022	more than 31 arguments in a call to identifier
C4023	assignment between *type1* and *type2* is compiler-dependent
C4024	*identifier* used in a conditional expression
C4026	conversion from *type1* to *type2* is compiler-dependent
C4027	*type* used as an lvalue
C4028	conversion from *type1* to *type2* is undefined
C4029	more than 511 external identifiers
C4030	initializer exceeds bit-field width
C4031	missing ’ in preprocessor line
C4033	unrecognized control line
C4034	more than 509 characters in a string literal
C4035	string/character literal contains non-portable characters
C4036	excess characters in multibyte character literal *token* ignored
C4037	overflow in constant *token*
C4039	overflow in hexadecimal escape sequence
C4041	unrecognized character escape sequence *character*
C4042	overflow in constant expression
C4043	result of unsigned comparison is constant

C4044	shifting a type by *number* bits is undefined
C4045	unreachable code
C4046	more than 15 levels of nested statements
C4047	switch statement with no cases
C4048	more than 257 cases in a switch
C4049	switch generates a huge table
C4050	pointer to a parameter/local *identifier* is an illegal return value
C4051	source code specifies an infinite loop
C4052	more than 127 identifiers declared in a block
C4053	reference to *type* elided
C4054	reference to **volatile** *type* elided
C4055	declaring type array of *type* is undefined
C4056	qualified function type ignored
C4057	unnamed *operator* in prototype

致命码

C6001	function not supported yet

盛群半导体股份有限公司（总公司）
台湾新竹市科学工业园区研新二路 3 号
电话: 886-3-563-1999
传真: 886-3-563-1189
网站: www.holtek.com.tw

盛群半导体股份有限公司（台北业务处）
台湾台北市南港区园区街 3 之 2 号 4 楼之 2
电话: 886-2-2655-7070
传真: 886-2-2655-7373
传真: 886-2-2655-7383 (International sales hotline)

盛扬半导体有限公司（上海业务处）
上海宜山路 889 号 2 号楼 7 楼　　200233
电话: 021-6485-5560
传真: 021-6485-0313
网站: www.holtek.com.cn

盛扬半导体有限公司（深圳业务处）
深圳市深南中路赛格广场 43 楼　　518031
电话: 0755-8346-5589
传真: 0755-8346-5590
ISDN: 0755-8346-5591

盛扬半导体有限公司（北京业务处）
北京市西城区宣武门西大街甲129号金隅大厦1721室100031
电话：010-6641-0030，6441-7751，6441-7752
传真：010-6641-0125

Holmate Semiconductor, Inc.（北美业务处）
46712 Fremont Blvd., Fremont, CA 94538
电话：510-252-9880
传真：510-252-9885
网站: www.holmate.com

Copyright © 2003 by HOLTEK SEMICONDUCTOR INC.

使用指南中所出现的信息在出版当时相信是正确的，然而盛群对于说明书的使用不负任何责任。文中提到的应用目的仅仅是用来做说明，盛群不保证或表示这些没有进一步修改的应用将是适当的，也不推荐它的产品使用在会由于故障或其它原因可能会对人身造成危害的应用。盛群产品不授权使用于救生、维生器件或系统中做为关键器件。盛群拥有不事先通知而修改产品的权利，对于最新的信息，请参考我们的网址 http://www.holtek.com.tw